VISUVAL PROGRAMMING (DSCSC32)

(BSC COMPUTER SCIENCE-IV)

ACHARYA NAGARJUNA UNIVERSITY

CENTRE FOR DISTANCE EDUCATION

NAGARJUNA NAGAR,

GUNTUR

ANDHRA PRADESH

CONTENTS

UNITS		PAGE NOS.
		1 – 62
UNIT – I	Getting Started with Visual C++	
	Dialogs and Controls	
	Messages and Commands	
		63 – 121
UNIT – II	Documents and Views	
	Drawing on the Screen	
	Printing and Print Preview	
		122 – 169
UNIT – III	Persistence and File I/O	
	Building a Complete Application: Showstring	
		170 - 229
UNIT – IV	Status Bars and Toolbars	
	Common Controls	
		230 – 286
UNIT – V	ActiveX Concepts	
	Building an ActiveX Container Application	

UNIT – I

Getting Started with Visual C++

In This Chapter

Creating a Windows Application
Creating a Dialog-Based Application
Creating DLLs, Console Applications, and More
Changing Your AppWizard Decisions
Understanding AppWizard's Code
Understanding a Multiple Document Interface Application
Understanding the Components of a Dialog-Based Application
Reviewing AppWizard Decisions and This Chapter

Creating a Windows Application

Visual C++ doesn't just compile code; it generates code. You can create a Windows application in minutes with a tool called AppWizard. In this chapter you'll learn how to tell AppWizard to make you a starter app with all the Windows boilerplate code you want. AppWizard is a very effective tool. It copies into your application the code that almost all Windows applications require. After all, you aren't the first programmer to need an application with resizable edges, minimize and maximize buttons, a File menu with Open, Close, Print Setup, Print, and Exit options, are you? AppWizard can make many kinds of applications, but what most people want, at least at first, is an executable (.exe) program. Most people also want AppWizard to produce boilerplate code—the classes, objects, and functions that have to be in every program. To create a program like this, Choose File, New and click the Projects tab in the New dialog box, as shown in Figure 1.1.

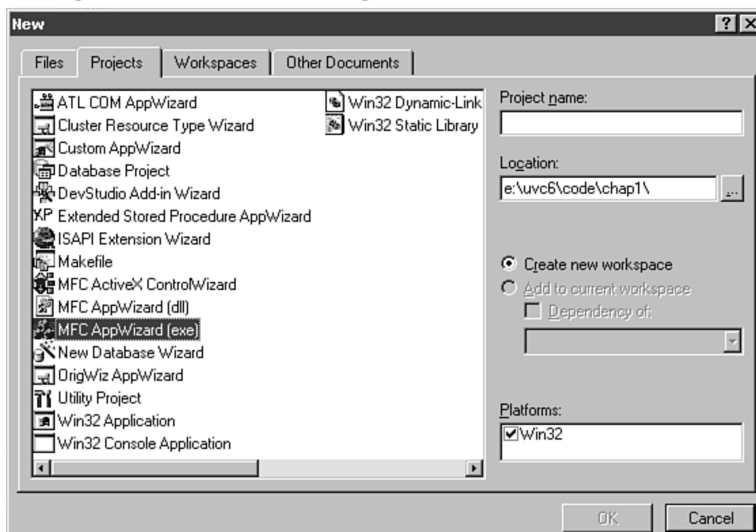


FIG. 1.1 The Projects tab of the New dialog box is where you choose the kind of application you want to build.

Choose MFC AppWizard (EXE) from the list box on the left, fill in a project name, and click OK. AppWizard will work through a number of steps. At each step, you make a decision about what kind of application you want and then click Next. At any time, you can click Back to return to a previous decision, Cancel to abandon the whole process, Help for more details, or Finish to skip to the end and create the application without answering any more questions (not recommended before the last step).

Deciding How Many Documents the Application Supports

The first decision to communicate to AppWizard, as shown in Figure 1.2, is whether your application should be MDI, SDI, or dialog based. AppWizard generates different code and classes for each of these application types.

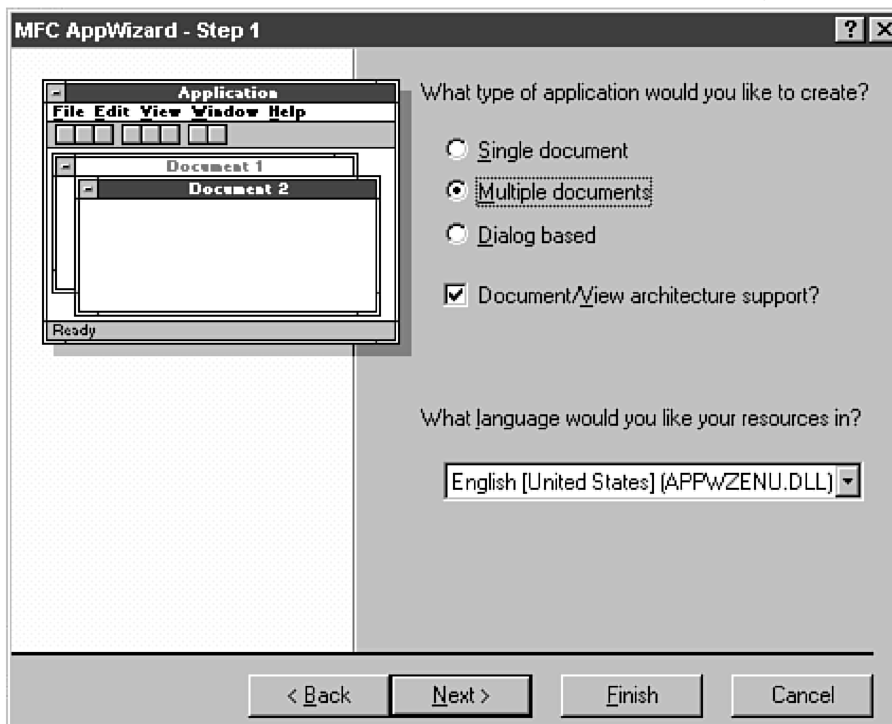


FIG. 1.2 The first step in building a typical application with AppWizard is choosing the interface.

The three application types to choose from are as follows:

- A *single document interface* (SDI) application, such as Notepad, has only one document open at a time. When you choose File, Open, the currently open file is closed before the new one is opened.
- A *multiple document interface* (MDI) application, such as Excel or Word, can open many documents (typically files) at once. There is a Window menu and a Close item on the File menu. It's a quirk of MFC that if you like multiple views on a single document, you must build an MDI application.
- A *dialog-based* application, such as the Character Map utility that comes with Windows and is shown in Figure 1.3, does not have a document at all. There are no menus. (If you'd like to see Character Map in action, it's usually in the

Accessories folder, reached by clicking Start. You may need to install it by using Add/Remove programs under Control Panel.)

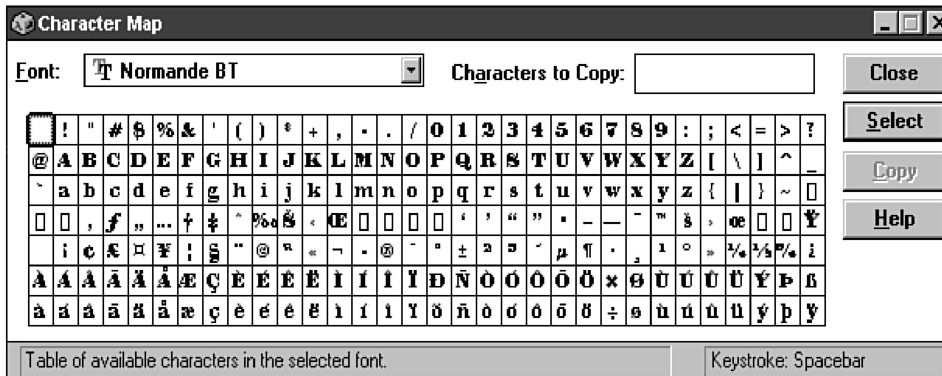


FIG. 1.3 Character Map is a dialog-based application

As you change the radio button selection, the picture on the left of the screen changes to demonstrate how the application appears if you choose this type of application. Beneath these choices is a checkbox for you to indicate whether you want support for the Document/View architecture. This framework for your applications is explained in Chapter 4, “Documents and Views.” Experienced Visual C++ developers, especially those who are porting an application from another development system, might choose to turn off this support. You should leave the option selected. Lower on the screen is a drop-down box to select the language for your resources. If you have set your system language to anything other than the default, English[United States], make sure you set your resources to that language, too. If you don’t, you will encounter unexpected behavior from ClassWizard later. (Of course, if your application is for users who will have their language set to U.S. English, you might not have a choice. In that case, change your system language under Control Panel.) Click Next after you make your choices.

Databases

The second step in creating an executable Windows program with AppWizard is to choose the level of database support, as shown in Figure 1.4.

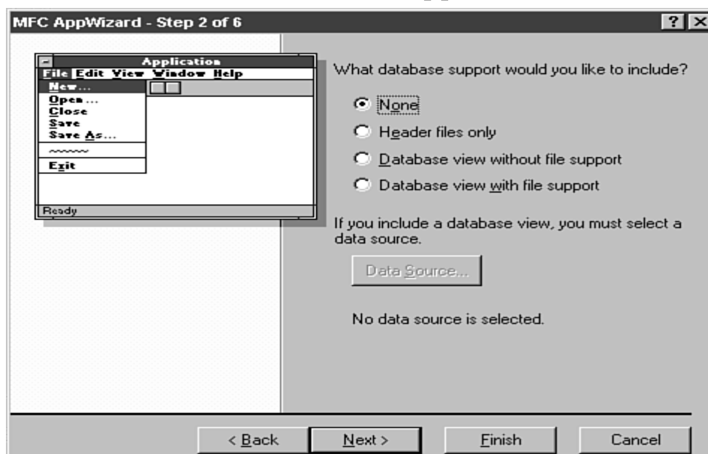


FIG. 1.4 The second step to building a typical application with AppWizard is to set the database options you will use.

There are four choices for database support:

- If you aren't writing a database application, choose None.
- If you want to have access to a database but don't want to derive your view from

  CFormView or have a Record menu, choose Header Files Only.
- If you want to derive your view from CFormView and have a Record menu but don't need to serialize a document, choose Database View Without File Support. You can update database records with CRecordset, an MFC class discussed in more detail in Chapter"Database Access."
- If you want to support databases as in the previous option but also need to save a document on disk (perhaps some user options), choose Database View With File Support.

If you choose to have a database view, you must specify a data source now. Click the Data Source button to set this up. As you select different radio buttons, the picture on the left changes to show you the results of your choice. Click Next to move to the next step.

Compound Document Support

The third step in running AppWizard to create an executable Windows program is to decide on the amount of compound document support you want to include, as shown in Figure 1.5. OLE (object linking and embedding) has been officially renamed ActiveX to clarify the recent technology shifts, most of which are hidden from you by MFC. ActiveX and OLE technology are jointly referred to as *compound document technology*. Chapter 13, "ActiveX Concepts," covers this technology in detail.

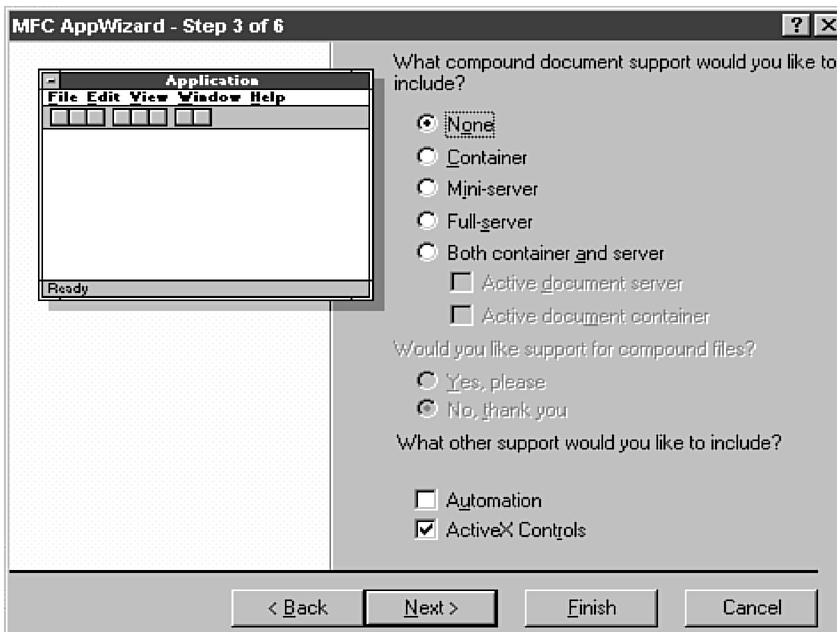


FIG. 1.5 The third step of building a typical application with AppWizard is to set the compound document support you w ill need.

There are five choices for compound document support:

- If you are not writing an ActiveX application, choose None.
- If you want your application to contain embedded or linked ActiveX objects, such as Word documents or Excel worksheets, choose Container. You learn to build an ActiveX container in Chapter 14, "Building an ActiveX Container Application."
- If you want your application to serve objects that can be embedded in other applications, but it never needs to run as a standalone application, choose Mini Server.
- If your application serves documents and also functions as a standalone application, choose Full Server. In Chapter 15, "Building an ActiveX Server Application," you learn to build an ActiveX full server.
- If you want your application to have the capability to contain objects from other applications and also to serve its objects to other applications, choose Both Container and Server.

If you choose to support compound documents, you can also support *compound files*. Compound files contain one or more ActiveX objects and are saved in a special way so that one of the objects can be changed without rewriting the whole file. This spares you a great deal of time. Use the radio buttons in the middle of this Step 3 dialog box to say Yes, Please, or No, Thank You to compound files. If you want your application to surrender control to other applications through automation, check the Automation check box. (Automation is the subject of Chapter 16, "Building an Automation Server.") If you want your application to use ActiveX controls, select the ActiveX Controls check box. Click Next to move to the next step.

Appearance and Other Options

The fourth step in running AppWizard to create an executable Windows program (see Figure 1.6) is to determine some of the interface appearance options for your application. This Step 4 dialog box contains a number of independent check boxes. Check them if you want a feature; leave them unchecked if you don't.

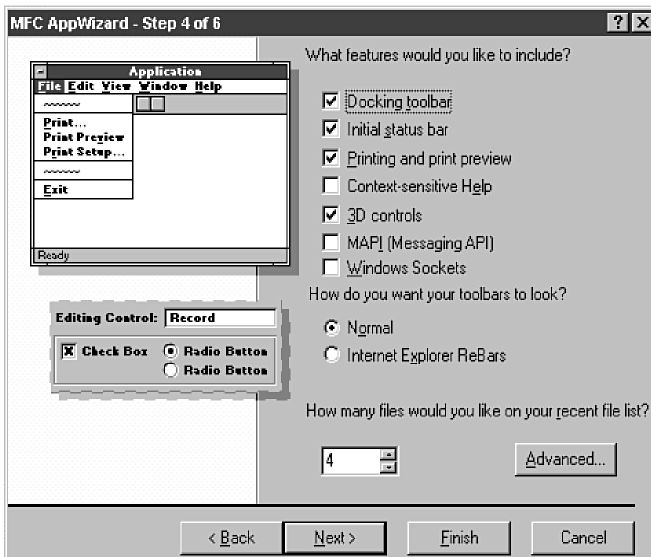


FIG. 1.6 The fourth step of building a typical application with AppWizard is to set some interface options.

The following are the options that affect your interface's appearance:

- *Docking Toolbar*. AppWizard sets up a toolbar for you. You can edit it to remove unwanted buttons or to add new ones linked to your own menu items. This is described in Chapter 9, "Status Bars and Toolbars."
- *Initial Status Bar*. AppWizard creates a status bar to display menu prompts and other messages. Later, you can write code to add indicators and other elements to this bar, as described in Chapter 9.
- *Printing and Print Preview*. Your application will have Print and Print Preview options on the File menu, and much of the code you need in order to implement printing will be generated by AppWizard. Chapter 6, "Printing and Print Preview," discusses the rest.
- *Context-Sensitive Help*. Your Help menu will gain Index and Using Help options, and some of the code needed to implement Help will be provided by AppWizard. This decision is hard to change later because quite a lot of code is added in different places when implementing Context-Sensitive Help. Chapter 11, "Help," describes Help implementation.
- *3D Controls*. Your application will look like a typical Windows 95 application. If you don't select this option, your dialog boxes will have a white background, and there will be no shadows around the edges of edit boxes, check boxes, and other controls.
- *MAPI(Messaging API)*. Your application will be able to use the Messaging API to send fax, email, or other messages. Chapter 18, "Sockets, MAPI, and the Internet," discusses the Messaging API.
- *Windows Sockets*. Your application can access the Internet directly, using protocols like FTP and HTTP (the World Wide Web protocol). Chapter 18 discusses sockets. You can produce Internet programs without enabling socket support if you use the new WinInet classes, discussed in Chapter 19, "Internet Programming with the WinInet Classes."

You can ask AppWizard to build applications with "traditional" toolbars, like those in Word or Visual C++ itself, or with toolbars like those in Internet Explorer. You can read more about this in Chapter 9. You can also set how many files you want to appear on the recent file list for this application. Four is the standard number; change it only if you have good reason to do so.

Clicking the Advanced button at the bottom of this Step 4 dialog box brings up the Advanced Options dialog box, which has two tabs. The Document Template Strings tab is shown in Figure 1.7. AppWizard builds many names and prompts from the name of your application, and sometimes it needs to abbreviate your application name. Until you are familiar with the names AppWizard builds, you should check them on this Document Template Strings dialog box and adjust them, if necessary. You can also change the mainframe caption, which appears in the title bar of your application. The file extension, if you choose one, will be incorporated into filenames saved by your application and will restrict the files initially displayed when the user chooses File, Open.

The Window Styles tab is shown in Figure 1.8. Here you can change the appearance of your application quite dramatically. The first check box, Use Split Window, adds all the code needed to implement splitter windows like those in the code editor of Developer Studio. The remainder of the Window Styles dialog box sets the appearance of your *main frame* and, for an MDI application, of your *MDI child frames*. Frames hold windows; the system menu, title bar, minimize and maximize boxes, and window edges are all frame properties. The main frame holds your entire application. An MDI application has a number of MDI child frames—one for each document window, inside the main frame.

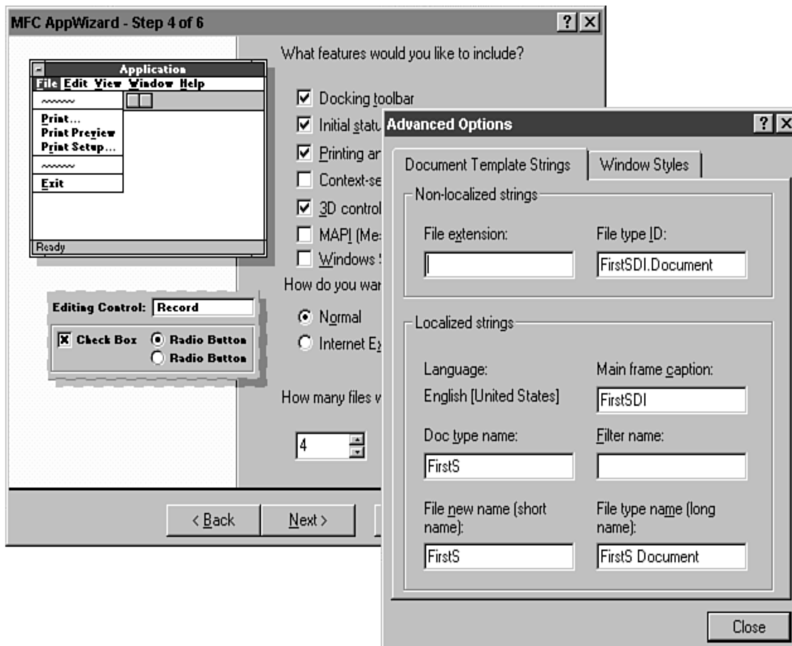


FIG. 1.7 The Document Template Strings tab of the Advanced Options dialog box lets you adjust the way names are abbreviated.

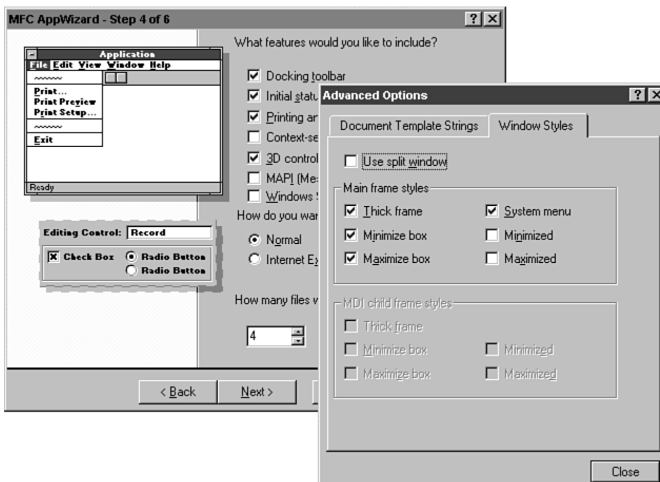


FIG. 1.8 The Window Styles tab of the Advanced Options dialog box lets you adjust the appearance of your windows.

Here are the properties you can set for frames:

- *Thick Frame*. The frame has a visibly thick edge and can be resized in the usual Windows way. Uncheck this to prevent resizing.
- *Minimize Box*. The frame has a minimize box in the top-right corner.
- *Maximize Box*. The frame has a maximize box in the top-right corner.
- *System Menu*. The frame has a system menu in the top-left corner.
- *Minimized*. The frame is minimized when the application starts. For SDI applications, this option will be ignored when the application is running under Windows 95.
- *Maximized*. The frame is maximized when the application starts. For SDI applications, this option will be ignored when the application is running under Windows 95.

When you have made your selections, click Close to return to step 4 and click Next to move on to the next step.

Other Options

The fifth step in running AppWizard to create an executable Windows program (see Figure 1.9) asks the leftover questions that are unrelated to menus, OLE, database access, or appearance. Do you want comments inserted in your code? You certainly do. That one is easy.

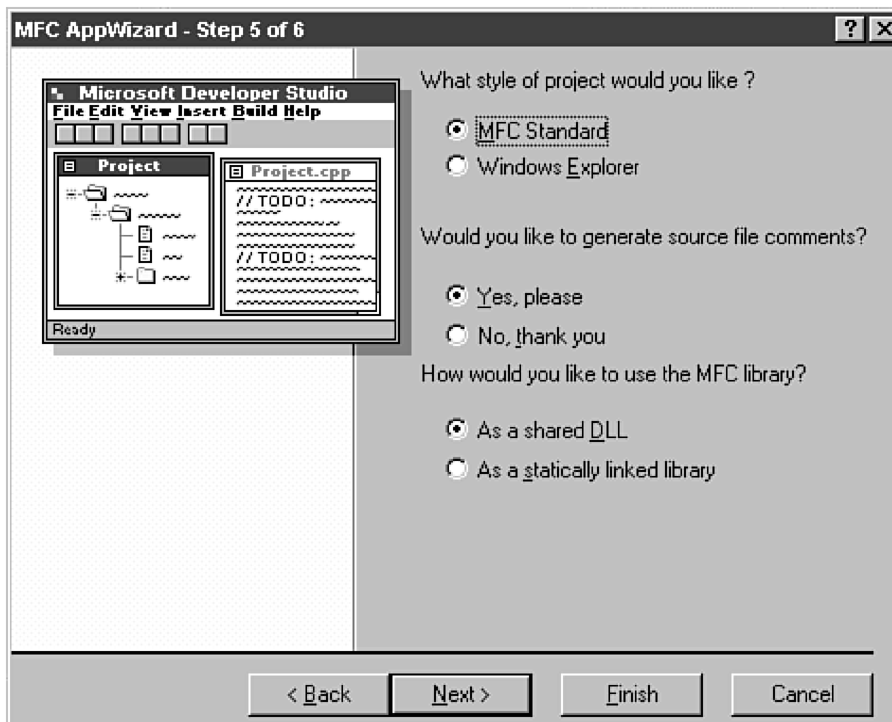


FIG. 1.9 The fifth step of building an application with AppWizard is to decide on comments and the MFC library.

The next question isn't as straightforward. Do you want the MFC library as a shared DLL or statically linked? A *DLL* (dynamic link library) is a collection of functions used by many different applications. Using a DLL makes your programs smaller but makes the installation a little more complex. Have you ever moved an executable to another directory, or another computer, only to find it won't run anymore because it's missing DLLs? If you statically link the MFC library into your application, it is larger, but it is easier to move and copy around. If your users are likely to be developers themselves and own at least one other application that uses the MFC DLL or aren't intimidated by the need to install DLLs as well as the program itself, choose the shared DLL option. The smaller executable is convenient for all. If your users are not developers, choose the statically linked option. It reduces the technical support issues you have to face with inexperienced users. If you write a good install program, you can feel more confident about using shared DLLs. After you've made your Step 5 choices, click Next to move to Step 6.

Filenames and Classnames

The final step in running AppWizard to create an executable Windows program is to confirm the classnames and the filenames that AppWizard creates for you, as shown in Figure 1.10. AppWizard uses the name of the project (FirstSDI in this example) to build the classnames and filenames. You should not need to change these names. If your application includes a view class, you can change the class from which it inherits; the default is CView, but many developers prefer to use another view, such as CScrollView or CEditView. The view classes are discussed in Chapter 4. Click Finish when this Step 6 dialog box is complete.

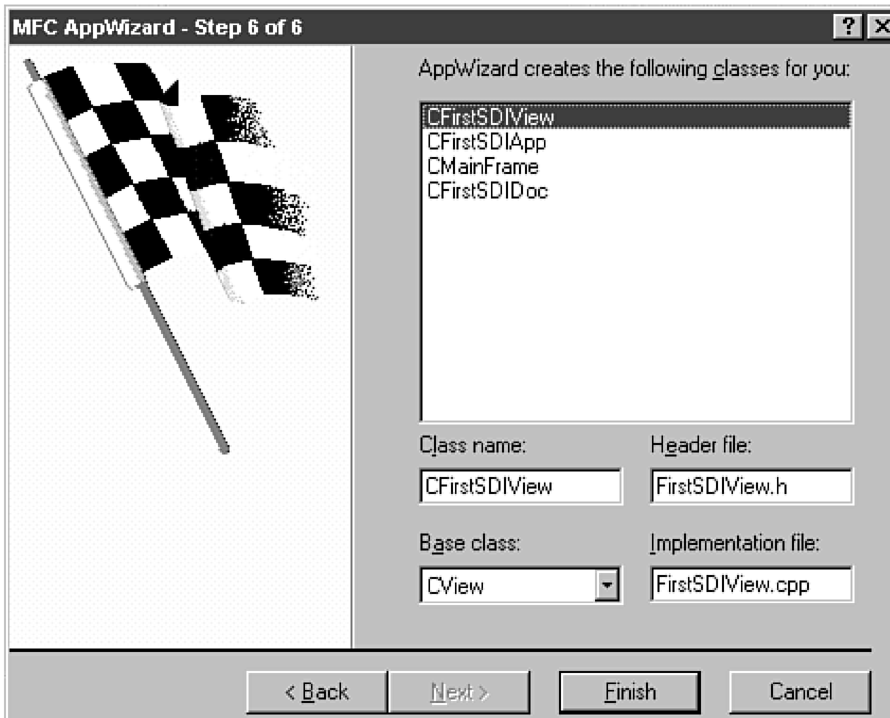


FIG. 1.10 The final step of building a typical application with AppWizard is to confirm filenames and classnames.

Creating the Application

After you click Finish, AppWizard shows you what is going to be created in a dialog box, similar to Figure 1.11. If anything here is wrong, click Cancel and work your way back through AppWizard with the Back buttons until you reach the dialog box you need to change. Move forward with Next, Finish; review this dialog box again; and click OK to actually create the application. This takes a few minutes, which is hardly surprising because hundreds of code lines, menus, dialog boxes, help text, and bitmaps are being generated for you in as many as 20 files. Let it work.

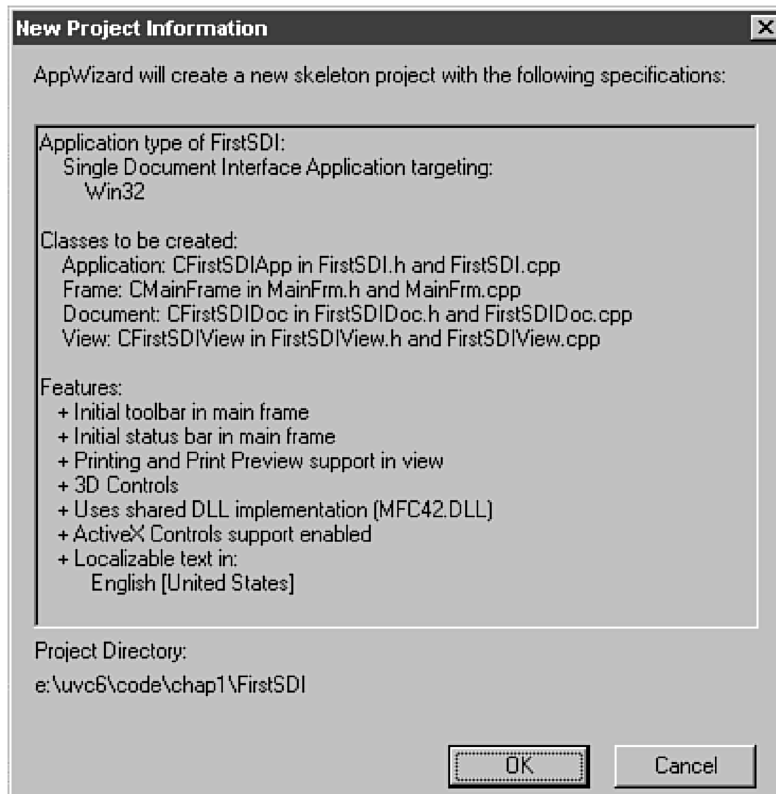


FIG. 1.11 When AppWizard is ready to build your application, you get one more chance to confirm everything.

Try It Yourself

If you haven't started Developer Studio already, do so now. If you've never used it before, you may find the interface intimidating. There's a full explanation of all the areas, toolbars, menus, and shortcuts in Appendix C, "The Visual Studio User Interface, Menus, and Toolbars." Bring up AppWizard by choosing File, New and clicking the Projects tab. On the Projects tab, fill in a folder name where you would like to keep your applications; AppWizard will make a new folder for each project. Fill in **FirstSDI** for the project name; then move through the six AppWizard steps. Choose an SDI application at Step 1, and on all the other steps simply leave the selections as they are and click Next. When AppWizard has created the project, choose Build, Build from the Developer Studio menu to compile and link the code. When the build is

complete, choose Build, Execute. You have a real, working Windows application, shown in Figure 1.12. Play around with it a little: Resize it, minimize it, maximize it.

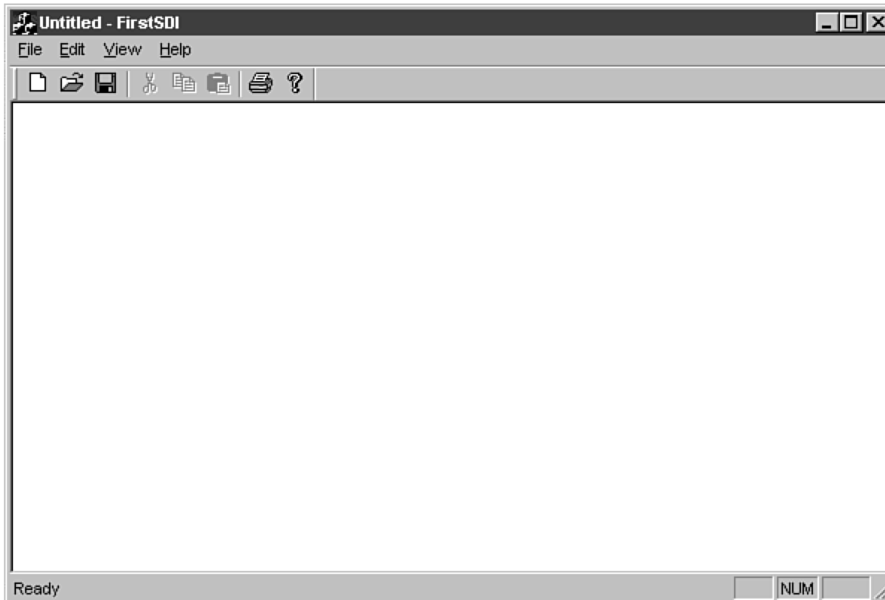


FIG. 1.12 Your first application looks like any fullfledged Windows application.

Try out the File menu by choosing File, Open; bring up the familiar Windows File Open dialog (though no matter what file you choose, nothing seems to happen); and then choose File, Exit to close the application. Execute the program again to continue exploring the capabilities that have been automatically generated for you. Move the mouse cursor over one of the toolbar buttons and pause; a ToolTip will appear, reminding you of the toolbar button's purpose. Click the Open button to confirm that it is connected to the File Open command you chose earlier. Open the View menu and click Toolbar to hide the toolbar; then choose View Toolbar again to restore it. Do the same thing with the status bar. Choose Help, About, and you'll see it even has an About box with its own name and the current year in the copyright date (see Figure 1.13).

Repeat these steps to create an MDI application called *FirstMDI*. The creation process will differ only on Step 0, where you specify the project name, and Step 1, where you choose an MDI application. Accept the defaults on all the other steps, create the application, build it, and execute it. You'll see something similar to Figure 1.14, an MDI application with a single document open. Try out the same operations you tried with FirstSDI. FIG. 1.12. Your first application looks like any fullfledged Windows application. Creating a Windows Application

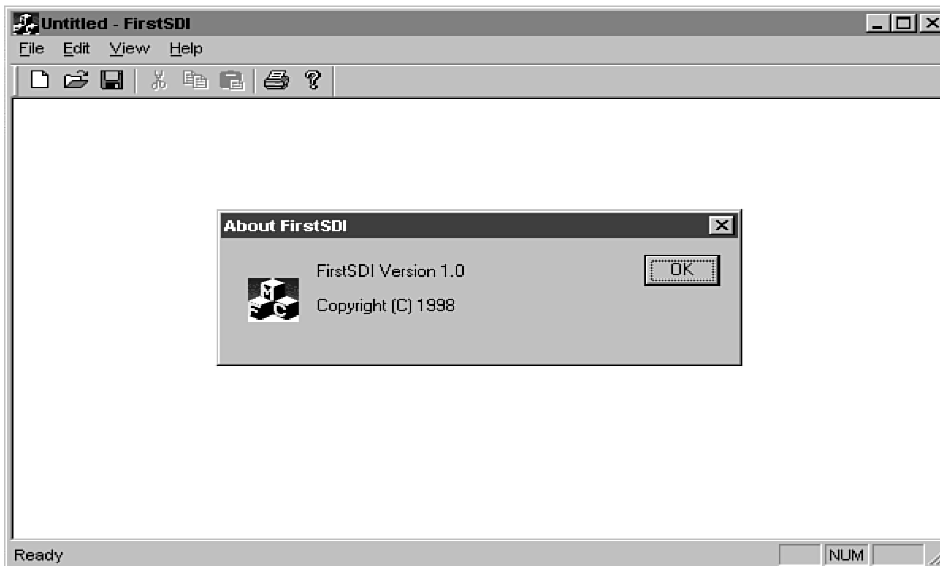


FIG. 1.13 You even get an About box in this start application.

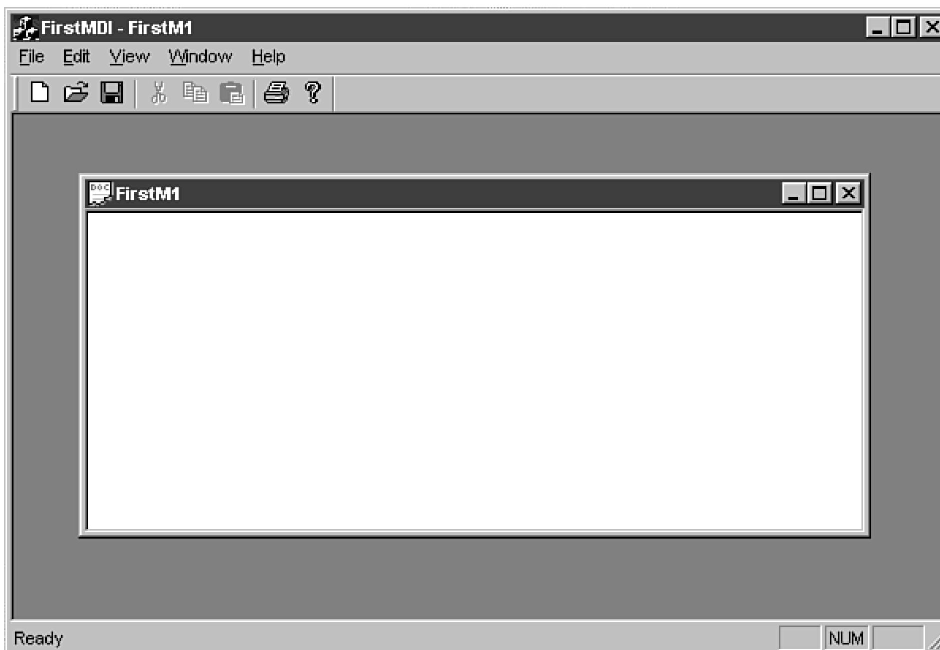


FIG. 1.14 An MDI application can display a number of documents at once.

Choose File, New, and a second window, FirstM2, appears. Try minimizing, maximizing, and restoring these windows. Switch among them using the Window menu. All this functionality is yours from AppWizard, and you don't have to write a single line of code to get it.

Creating a Dialog-Based Application

A dialog-based application has no menus other than the system menu, and it cannot save or open a file. This makes it good for simple utilities like the Windows Character Map. The AppWizard process is a little different for a dialog-based

application, primarily because such applications can't have a document and therefore can't support database access or compound documents. To create a dialog-based application, start AppWizard as you did for the SDI or MDI application, but in Step 1 choose a dialog-based application, as shown in Figure 1.15. Call this application **FirstDialog**.

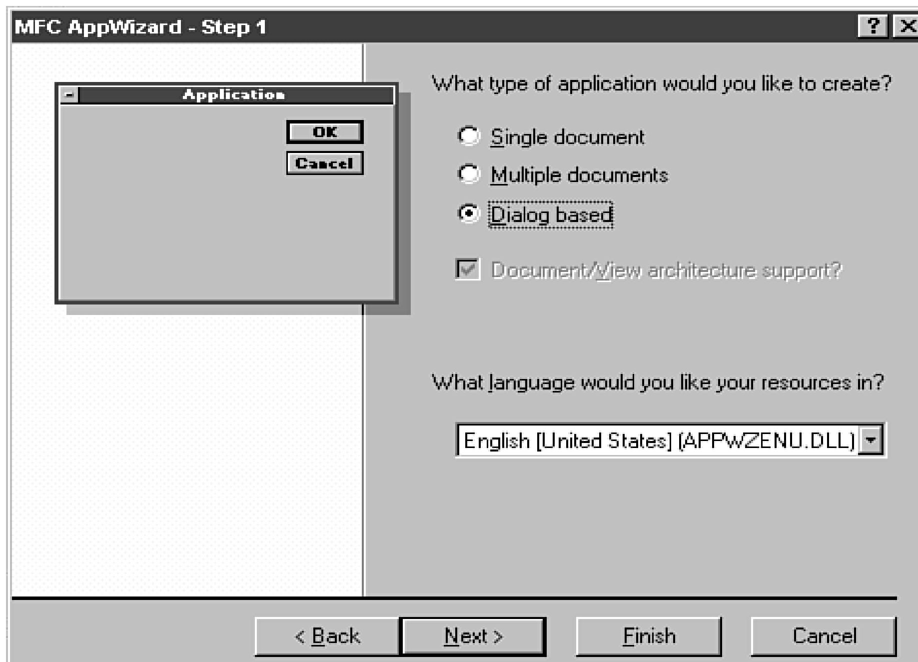


FIG. 1.15 To create a dialogbased application, specify your preference in Step 1 of the AppWizard process.

Choose Dialog Based and click Next to move to Step 2, shown in Figure 1.16.

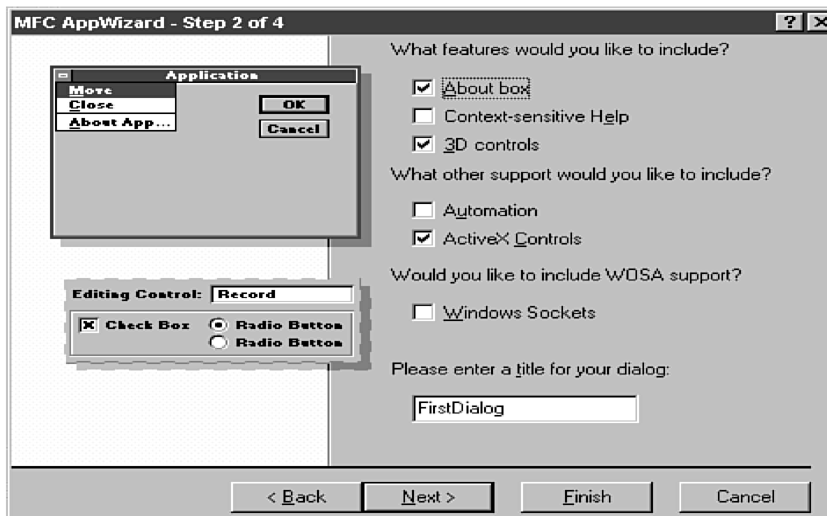


FIG. 1.16 Step 2 of the AppWizard process for a dialog-based application involves choosing Help, Automation, ActiveX, and Sockets settings.

If you would like an About item on the system menu, select the About Box item. To have AppWizard lay the framework for Help, select the Context-Sensitive Help option. The third check box, 3D Controls, should be selected for most Windows 95 and Windows NT applications. If you want your application to surrender control to other applications through automation, as discussed in Chapter 16, select the Automation check box. If you want your application to contain ActiveX controls, select the ActiveX Controls check box. If you are planning to have this application work over the Internet with sockets, check the Windows Sockets box. (Dialog-based apps can't use MAPI because they have no document.) Click Next to move to the third step, shown in Figure 1.17.

As with the SDI and MDI applications created earlier, you want comments in your code. The decision between static linking and a shared DLL is also the same as for the SDI and MDI applications. If your users are likely to already have the MFC DLLs (because they are developers or because they have another product that uses the DLL) or if they won't mind installing the DLLs as well as your executable, go with the shared DLL to make a smaller executable file and a faster link. Otherwise, choose As A Statically Linked Library. Click Next to move to the final step, shown in Figure 1.18.

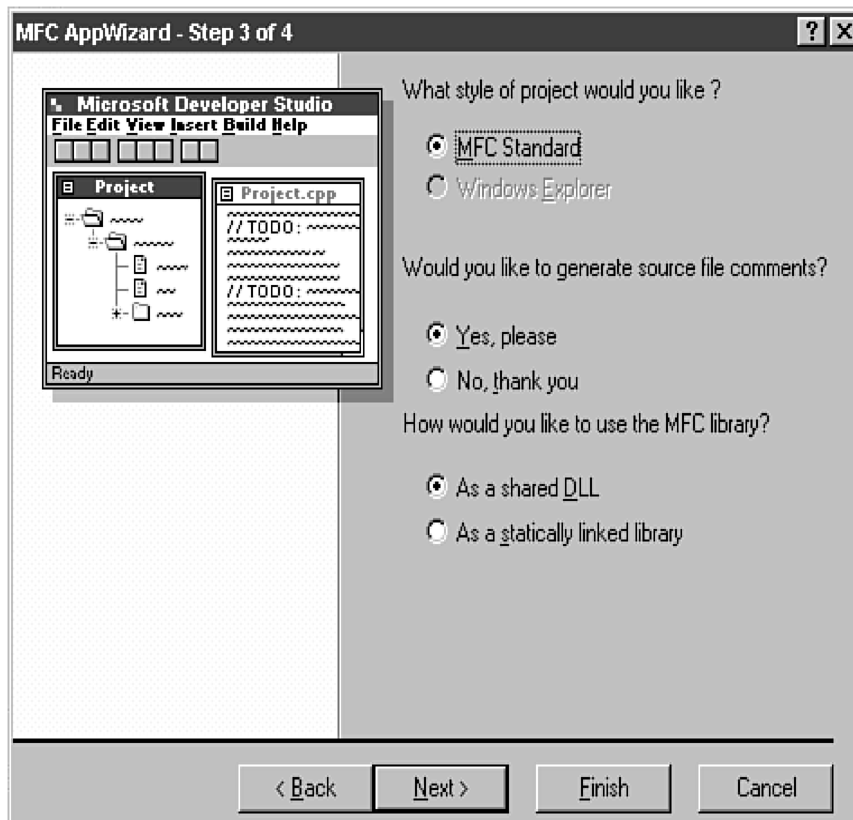


FIG. 1.17 Step 3 of the AppWizard process for a dialogbased application deals with comments and the MFC library.

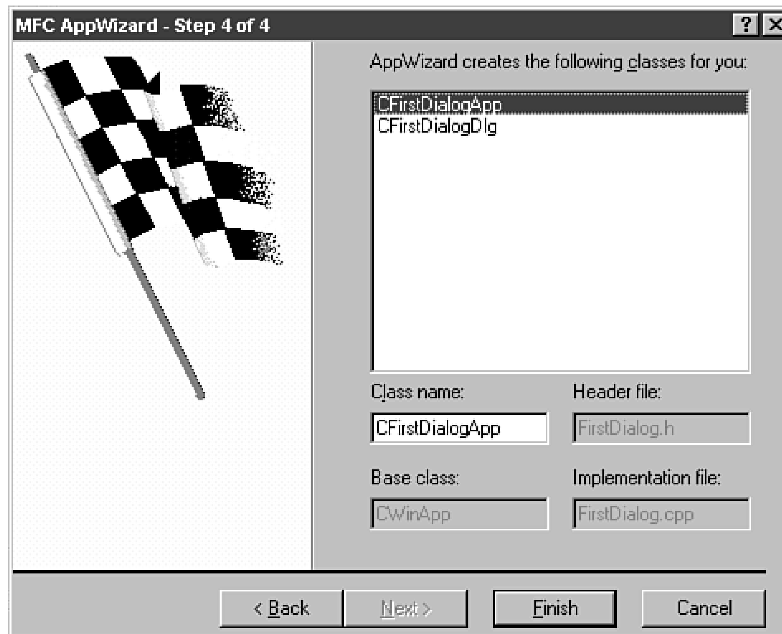


FIG. 1.18 Step 4 of the AppWizard process for a dialogbased application gives you a chance to adjust filenames and classnames.

In this step you can change the names AppWizard chooses for files and classes. This is rarely a good idea because it will confuse people who maintain your code if the filenames can't be easily distinguished from the classnames, and vice versa. If you realize after looking at this dialog that you made a poor choice of project name, use Back to move all the way back to the New Project Workspace dialog, change the name, click Create, and then use Next to return to this dialog. Click Finish to see the summary of the files and classes to be created, similar to that in Figure 1.19.

If any information on this dialog isn't what you wanted, click Cancel and then use Back to move to the appropriate step and change your choices. When the information is right, click OK and watch as the application is created. To try it yourself, create an empty dialog-based application yourself, call it *FirstDialog*, and accept the defaults for each step of AppWizard. When it's complete, choose Build, Build to compile and link the application. Choose Build, Execute to see it in action. Figure 1.20 shows the empty dialog-based application running.

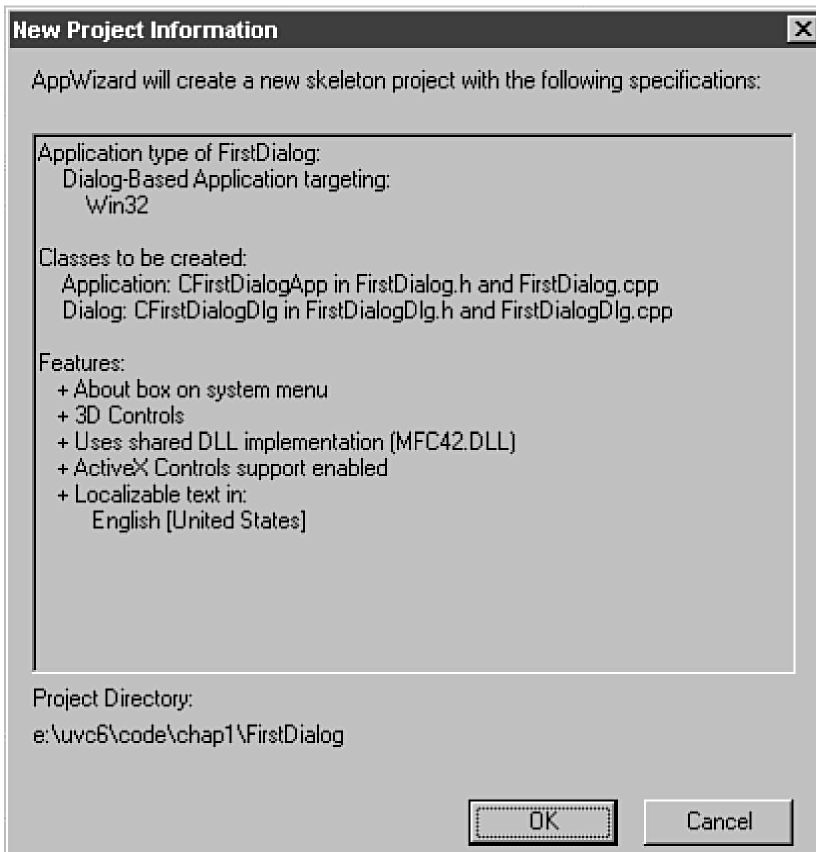


FIG. 1.19 AppWizard confirms the files and classes before creating them.

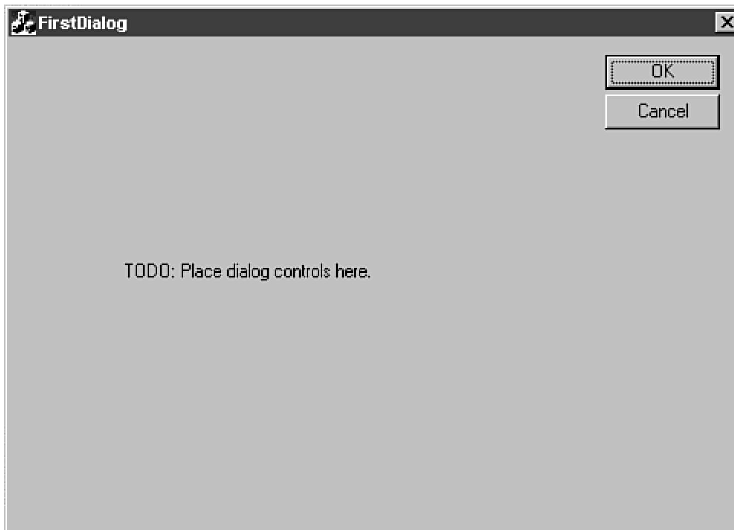


FIG. 1.20 A starter dialog application includes a reminder of the work ahead of you.

Clicking the OK or Cancel button, or the X in the top-right corner, makes the dialog disappear. Clicking the system menu in the top-left corner gives you a choice of Move, Close, or About. Figure 1.21 shows the About box that was generated for you.

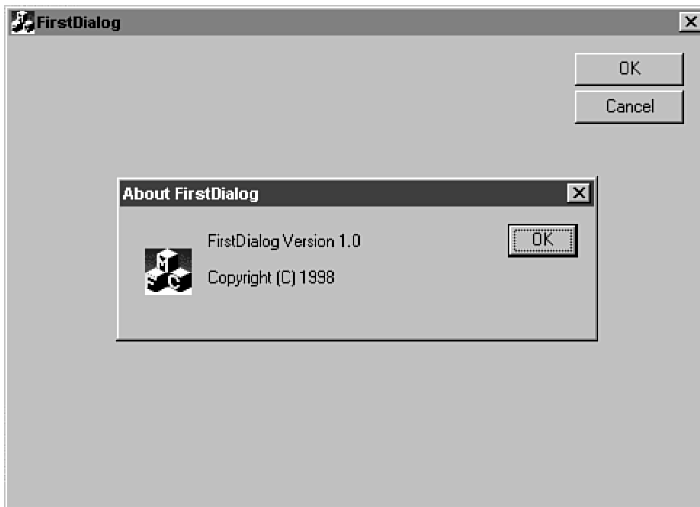


FIG. 1.21 The same About box is generated for SDI, MDI, and dialog-based applications

Creating DLLs, Console Applications, and More

Although most people use AppWizard to create an executable program, it can make many other kinds of projects. You choose File, New and then the Projects tab, as discussed at the start of this chapter, but choose a different wizard from the list on the left of the New dialog box, shown in Figure 1.1. The following are some of the other projects AppWizard can create:

- ATL COM AppWizard
- Custom AppWizard
- Database Project
- DevStudio Add-In Wizard
- Extended Stored Procedure AppWizard
- ISAPI Extension Wizard
- Makefile
- MFC ActiveX ControlWizard
- MFC AppWizard (dll)
- Utility Project
- Win32 Application
- Win32 Console Application
- Win32 Dynamic Link Library
- Win32 Static Library

These projects are explained in the following sections.

ATL COM AppWizard

ATL is the Active Template Library, and it's used to write small ActiveX controls. It's generally used by developers who have already mastered writing MFC ActiveX controls, though an MFC background is not required to learn ATL. Chapter 17 introduces important control concepts while demonstrating how to build an MFC control; Chapter 21, "The Active Template Library," teaches you ATL.

Custom AppWizard

Perhaps you work in a large programming shop that builds many applications. Although AppWizard saves a lot of time, your programmers may spend a day or two at the start of each project pasting in your own *boilerplate*, which is material that is the same in every one of your projects. You may find it well worth your time to build a Custom AppWizard, a wizard of your very own that puts in your boilerplate as well as the standard MFC material. After you have done this, your application type is added to the list box on the left of the Projects tab of the New dialog box shown in Figure 1.1. Creating and using Custom AppWizards is discussed in Chapter 25, "Achieving Reuse with the Gallery and Your Own AppWizards."

Database Project

If you have installed the Enterprise Edition of Visual C++, you can create a database project. This is discussed in Chapter 23, "SQL and the Enterprise Edition."

DevStudio Add-In Wizard

Add-ins are like macros that automate Developer Studio, but they are written in C++ or another programming language; macros are written in VBScript. They use automation to manipulate Developer Studio.

ISAPI Extension Wizard

ISAPI stands for Internet Server API and refers to functions you can call to interact with a running copy of Microsoft Internet Information Server, a World Wide Web server program that serves out Web pages in response to client requests. You can use this API to write DLLs used by programs that go far beyond browsing the Web to sophisticated automatic information retrieval. This process is discussed in Chapter 18.

Makefile

If you want to create a project that is used with a different make utility than Developer Studio, choose this wizard from the left list in the New Project Workspace dialog box. No code is generated. If you don't know what a make utility is, don't worry—this wizard is for those who prefer to use a standalone tool to replace one portion of Developer Studio.

MFC ActiveX ControlWizard

ActiveX controls are controls you write that can be used on a Visual C++ dialog, a Visual Basic form, or even a Web page. These controls are the 32-bit replacement for the VBX controls many developers were using to achieve intuitive interfaces or to avoid reinventing the wheel on every project. Chapter 17 guides you through building a control with this wizard.

MFC AppWizard (DLL)

If you want to collect a number of functions into a DLL, and these functions use MFC classes, choose this wizard. (If the functions don't use MFC, choose Win32 Dynamic Link Library, discussed a little later in this section.) Building a DLL is covered in Chapter 28, "Future Explorations." AppWizard generates code for you so you can get started.

Win32 Application

There are times when you want to create a Windows application in Visual C++ that doesn't use MFC and doesn't start with the boilerplate code that AppWizard produces for you. To create such an application, choose the Win32 Application wizard from the left list in the Projects tab, fill in the name and folder for your project, and click OK. You are not asked any questions; AppWizard simply creates a project file for you and opens it. You have to create all your code from scratch and insert the files into the project.

Changing Your AppWizard Decisions

Running AppWizard is a one-time task. Assuming you are making a typical application, you choose File, New; click the Projects tab; enter a name and folder; choose MFC Application (exe); go through the six steps; create the application starter files; and then never touch AppWizard again. However, what if you choose not to have online Help and later realize you should have included it?

AppWizard, despite the name, isn't really magic. It pastes in bits and pieces of code you need, and you can paste in those very same bits yourself. Here's how to find out what you need to paste in. First, create a project with the same options you used in creating the project whose settings you want to change, and don't add any code to it. Second, in a different folder create a project with the same name and all the same settings, except the one thing you want to change (Context-Sensitive Help in this example). Compare the files, using WinDiff, which comes with Visual C++. Now you know what bits and pieces you need to add to your full-of-code project to implement the feature you forgot to ask AppWizard for. Some developers, if they discover their mistake soon enough, find it quicker to create a new project with the desired features and then paste their own functions and resources from the partially built project into the new empty one. It's only a matter of taste, but after you go through either process for changing your mind, you probably will move a little more slowly through those AppWizard dialog boxes.

Understanding AppWizard's Code

The code generated by AppWizard may not make sense to you right away, especially if you haven't written a C++ program before. You don't need to understand this code in order to write your own simple applications. Your programs will be better ones, though, if you know what they are doing, so a quick tour of AppWizard's boilerplate code is a good idea. You'll see the core of an SDI application, an MDI application, and a dialog-based application. You'll need the starter applications FirstSDI, FirstMDI, and FirstDialog, so if you didn't create them earlier, do so now. If you're unfamiliar with the Developer Studio interface, glance through Appendix C to learn how to edit code and look at classes.

A Single Document Interface Application

An SDI application has menus that the user uses to open one document at a time and work with that document. This section presents the code that is generated when you create an SDI application with no database or compound document support, with a toolbar, a status bar, Help, 3D controls, source file comments, and with the MFC library as a shared DLL—in other words, when you accept all the AppWizard defaults after Step1. Five classes have been created for you. For the application FirstSDI, they are as follows:

- CAboutDlg, a dialog class for the About dialog box
- CFirstSDIApp, a CWinApp class for the entire application
- CFirstSDIDoc, a document class
- CFirstSDIView, a view class
- CMainFrame, a frame class

Dialog classes are discussed in Chapter 2, "Dialogs and Controls." Document, view, and frame classes are discussed in Chapter 4. The header file for CFirstSDIApp is shown in Listing 1.1. The easiest way for you to see this code is to double-click on the classname, CFirstDSIApp, in the ClassView pane. This will edit the header file for the class.

Listing 1.1 FirstSDI.h—Main Header File for the FirstSDI Application

```
// FirstSDI.h : main header file for the FIRSTSDI application
//
#if!defined(AFX_FIRSTSDI_H__CDF38D8A_8718_11D0_B02C_0080C81A3AA2__INCLU
DED_)
#define
AFX_FIRSTSDI_H__CDF38D8A_8718_11D0_B02C_0080C81A3AA2__INCLUDED_
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif
#include "resource.h" // main symbols

///////////////////////////////////////////////////////////////////////////
/
// CFirstSDIApp:
// See FirstSDI.cpp for the implementation of this class
//
class CFirstSDIApp : public CWinApp
{
public:
CFirstSDIApp();
// Overrides
```

```
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CFirstSDIApp)
public:
virtual BOOL InitInstance();
//}}AFX_VIRTUAL
// Implementation
//{{AFX_MSG(CFirstSDIApp)
afx_msg void OnAppAbout();
// NOTE - The ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
///////////////////////////////////////////////////////////////////
///////
//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations
// immediately before the previous line.
#endif
//!defined(AFX_FIRSTSDI_H__CDF38D8A_8718_11D0_B02C_0080C81A3AA2__INCLU
DED_)
```

This code is confusing at the beginning. The #if(!defined) followed by the very long string (yours will be different) is a clever form of include guarding. You may have seen a code snippet

like this before:

```
#ifndef test_h
#include "test.h"
#define test_h
#endif
```

This guarantees that the file test.h will never be included more than once. Including the same file more than once is quite likely in C++. Imagine that you define a class called Employee, and it uses a class called Manager. If the header files for both Employee and Manager include, for example, BigCorp.h, you will get error messages from the compiler about "redefining" the symbols in BigCorp.h the second time it is included.

There is a problem with this approach: If someone includes test.h but forgets to set test_h,

your code will include test.h the second time. The solution is to put the test and the definition in the header file instead, so that test.h looks like this:

```
#ifndef test_h
... the entire header file
#define test_h
#endif
```

All AppWizard did was generate a more complicated variable name than test_h (this wild name prevents problems when you have several files, in different folders and projects, with the same name) and use a slightly different syntax to check the variable. The #pragma once code is also designed to prevent multiple definitions if this file is ever included twice. The actual meat of the file is the definition of the class CFirstSDIApp. This class inherits from CWinApp, an MFC class that provides most of the functionality you need. AppWizard has generated some functions for this class that override the ones inherited from the base class. The section of code that begins //Overrides is for virtual function overrides. AppWizard generated the odd-looking comments that surround the declaration of InitInstance(): ClassWizard will use these to simplify the job of adding other overrides later, if they are necessary. The next section of code is a message map and declares there is a function called OnAppAbout. You can learn all about message maps in Chapter 3, "Messages and Commands." AppWizard generated the code for the CFirstSDIApp constructor, InitInstance(), and OnAppAbout() in the file firstsdi.cpp. Here's the constructor, which initializes a CFirstSDIApp object as it is created:

```
CFirstSDIApp::CFirstSDIApp()
{
// TODO: add construction code here,
// Place all significant initialization in InitInstance
}
```

This is a typical Microsoft constructor. Because constructors don't return values, there's no easy way to indicate that there has been a problem with the initialization. There are several ways to deal with this. Microsoft's approach is a two-stage initialization, with a separate initializing function so that construction does no initialization. For an application, that function is called

InitInstance(), shown in Listing 1.2.

Listing 1.2 CFirstSDIApp::InitInstance()

```
BOOL CFirstSDIApp::InitInstance()
{
AfxEnableControlContainer();
// Standard initialization
// If you are not using these features and want to reduce the size
// of your final executable, you should remove from the following
// the specific initialization routines you don't need.
#ifdef _AFXDLL
Enable3dControls(); // Call this when using MFC in a shared DLL
#else
Enable3dControlsStatic(); // Call this when linking to MFC statically
#endif
// Change the registry key under which our settings are stored.
// You should modify this string to be something appropriate,
// such as the name of your company or organization.
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
```

```
LoadStdProfileSettings(); // Load standard INI file options (including
// MRU)
// Register the application's document templates. Document templates
// serve as the connection between documents, frame windows, and views.
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
IDR_MAINFRAME,
RUNTIME_CLASS(CFirstSDIDoc),
RUNTIME_CLASS(CMainFrame), // main SDI frame window
RUNTIME_CLASS(CFirstSDIView));
AddDocTemplate(pDocTemplate);
// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
// Dispatch commands specified on the command line
if (!ProcessShellCommand(cmdInfo))
return FALSE;
// The one and only window has been initialized, so show and update it.
m_pMainWnd->ShowWindow(SW_SHOW);
m_pMainWnd->UpdateWindow();
return TRUE;
}
```

InitInstance gets applications ready to go. This one starts by enabling the application to contain ActiveX controls with a call to AfxEnableControlContainer() and then turns on 3D controls. It then sets up the Registry key under which this application will be registered. (The Registry is introduced in Chapter 7, "Persistence and File I/O." If you've never heard of it, you can ignore it for now.)

InitInstance() goes on to register single document templates, which is what makes this an

SDI application. Documents, views, frames, and document templates are all discussed in Chapter 4.

Following the comment about parsing the command line, InitInstance() sets up an empty CCommandLineInfo object to hold any parameters that may have been passed to the application when it was run, and it calls ParseCommandLine() to fill that. Finally, it calls ProcessShellCommand() to do whatever those parameters requested. This means your application can support command-line parameters to let users save time and effort, without effort on your part. For example, if the user types at the command line **FirstSDI fooble**, the application starts and opens the file called *fooble*. The command-line parameters that ProcessShellCommand() supports are the following:

Parameter	Action
None	Start app and open new file.
Filename	Start app and open file.
/p filename	Start app and print file to default printer.
/pt filename printerdriver port	Start app and print file to the specified printer.
/dde	Start app and await DDE command.
/Automation	Start app as an OLE automation server.
/Embedding	Start app to edit an embedded OLE item.

If you would like to implement other behavior, make a class that inherits from CCommandLineInfo to hold the parsed command line; then override CWinApp:: ParseCommandLine() and CWinApp::ProcessShellCommand() in your own App class.

That's the end of InitInstance(). It returns TRUE to indicate that the rest of the application should now run. The message map in the header file indicated that the function OnAppAbout() handles a message. Which one? Here's the message map from the source file:

```
BEGIN_MESSAGE_MAP(CFirstSDIApp, CWinApp)
//{{AFX_MSG_MAP(CFirstSDIApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
// NOTE - The ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
// Standard file-based document commands
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
// Standard print setup command
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

T I P

Understanding AppWizard's Code

This message map catches commands from menus, as discussed in Chapter 3. When the user chooses Help About, CFirstSDIApp::OnAppAbout() will be called. When the user chooses File New, File Open, or File Print Setup, functions from CWinApp will handle that work for you. (You would override those functions if you wanted to do something special for those menu choices.)

OnAppAbout() looks like this:

```
void CFirstSDIApp::OnAppAbout()
{
CAboutDlg aboutDlg;
aboutDlg.DoModal();
}
```

This code declares an object that is an instance of CAboutDlg, and calls its DoModal() function to display the dialog onscreen. (Dialog classes and the DoModal() function are both covered in Chapter 2.) There's no need to handle OK or Cancel in any special way—this is just an About box.

Other Files

If you selected Context-Sensitive Help, AppWizard generates an .HPJ file and a number of .RTF files to give some context-sensitive help. These files are discussed in Chapter 11 in the "Components of the Help System" section. AppWizard also generates a README.TXT file that explains what all the other files are and what classes have been created. Read this file if all the similar filenames become confusing. There are also a number of project files used to hold your settings and options, to speed build time by saving partial results, and to keep information about all your variables and functions. These files have extensions like .ncb, .aps, .dsw, and so on. You can safely ignore these files because you will not be using them directly.

Understanding a Multiple Document Interface Application

A multiple document interface application also has menus, and it enables the user to have more than one document open at once. This section presents the code that is generated when you choose an MDI application with no database or compound document support, but instead with a toolbar, a status bar, Help, 3D controls, source file comments, and the MFC library as a shared DLL. As with the SDI application, these are the defaults after Step 1. The focus here ison what differs from the SDI application in the previous section.

Five classes have been created for you. For the application FirstMDI, they are

- CAboutDlg, a dialog class for the About dialog box
- CFirstMDIApp, a CWinApp class for the entire application
- CFirstMDIDoc, a document class
- CFirstMDIView, a view class
- CMainFrame, a frame class

The App class header is shown in Listing 1.3.

Listing 1.3 FirstMDI.h—Main Header File for the FirstMDI Application

```
// FirstMDI.h : main header file for the FIRSTMDI application
//
#if!defined(AFX_FIRSTMDI_H__CDF38D9E_8718_11D0_B02C_0080C81A3AA2__INCLU    DED_)
#define
AFX_FIRSTMDI_H__CDF38D9E_8718_11D0_B02C_0080C81A3AA2__INCLUDED_
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif
```

```
#include "resource.h" // main symbols
//////////////////////////////////////////////////////////////////////////////
// CFirstMDIApp:
// See FirstMDI.cpp for the implementation of this class
//
class CFirstMDIApp : public CWinApp
{
public:
CFirstMDIApp();
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CFirstMDIApp)
public:
virtual BOOL InitInstance();
//}}AFX_VIRTUAL
// Implementation
//{{AFX_MSG(CFirstMDIApp)
afx_msg void OnAppAbout();
// NOTE - The ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
//////////////////////////////////////////////////////////////////////////////
//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately
// before the previous line.
#endif
//!defined(AFX_FIRSTMDI_H__CDF38D9E_8718_11D0_B02C_0080C81A3AA2__INCLUDED_)
```

How does this differ from FirstSDI.h? Only in the classnames. The constructor is also the sameas before. OnAppAbout() is just like the SDI version. How about InitInstance()? It is in Listing 1.4.

Listing 1.4 CFirstMDIApp::InitInstance()

```
BOOL CFirstMDIApp::InitInstance()
{
AfxEnableControlContainer();
// Standard initialization
```

```
// If you are not using these features and want to reduce the size
// of your final executable, you should remove from the following
// the specific initialization routines you don't need.
#ifdef _AFXDLL
Enable3dControls(); // Call this when using MFC in a shared DLL
#else
Enable3dControlsStatic(); // Call this when linking to MFC statically
#endif
// Change the registry key under which your settings are stored.
// You should modify this string to be something appropriate,
// such as the name of your company or organization.
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
LoadStdProfileSettings(); // Load standard INI file options (including
// MRU)
// Register the application's document templates. Document templates
// serve as the connection between documents, frame windows, and views.
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
IDR_FIRSTMTYPE,
RUNTIME_CLASS(CFirstMDIDoc),
RUNTIME_CLASS(CChildFrame), // custom MDI child frame
RUNTIME_CLASS(CFirstMDIView));
AddDocTemplate(pDocTemplate);
// create main MDI Frame window
CMainFrame* pMainFrame = new CMainFrame;
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
return FALSE;
m_pMainWnd = pMainFrame;
// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
// Dispatch commands specified on the command line
if (!ProcessShellCommand(cmdInfo))
return FALSE;
// The main window has been initialized, so show and update it.
pMainFrame->ShowWindow(m_nCmdShow);
pMainFrame->UpdateWindow();
return TRUE;
}
```

What's different here? Using WinDiff can help. WinDiff is a tool that comes with Visual C++ and is reached from the Tools menu. (If WinDiff isn't on your Tools menu,

see the “Tools” section of Appendix C.) Using WinDiff to compare the FirstSDI and FirstMDI versions of InitInstance() confirms that, other than the classnames, the differences are

- The MDI application sets up a CMultiDocTemplate and the SDI application sets up a CSingleDocTemplate, as discussed in Chapter 4.
- The MDI application sets up a mainframe window and then shows it; the SDI application does not.

This shows a major advantage of the Document/View paradigm: It enables an enormous design decision to affect only a small amount of the code in your project and hides that decision as much as possible.

Understanding the Components of a Dialog-Based Application

Dialog applications are much simpler than SDI and MDI applications. Create one called

FirstDialog, with an About box, no Help, 3D controls, no automation, ActiveX control support, no sockets, source file comments, and MFC as a shared DLL. In other words, accept all the default options.

Three classes have been created for you for the application called *FirstMDI*:

- CAboutDlg, a dialog class for the About dialog box
- CFirstDialogApp, a CWinApp class for the entire application
- CFirstDialogDlg, a dialog class for the entire application

The dialog classes are the subject of Chapter 2. Listing 1.5 shows the header file for CFirstDialogApp.

Listing 1.5 dialog16.h—Main Header File

```
// FirstDialog.h : main header file for the FIRSTDIALOG application
//
#if!defined(AFX_FIRSTDIALOG_H__CDF38DB4_8718_11D0_B02C_0080C81A3AA2__IN
CLUDED_)
#define
AFX_FIRSTDIALOG_H__CDF38DB4_8718_11D0_B02C_0080C81A3AA2__INCLUDED_
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
#ifndef __AFXWIN_H__
#error include ‘stdafx.h’ before including this file for PCH
#endif
#include “resource.h” // main symbols
//////////////////////////////////////////////////////////////////////
///////
// CFirstDialogApp:
// See FirstDialog.cpp for the implementation of this class
```

```
//
class CFirstDialogApp : public CWinApp
{
public:
CFirstDialogApp();
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CFirstDialogApp)
public:
virtual BOOL InitInstance();
//}}AFX_VIRTUAL
// Implementation
//{{AFX_MSG(CFirstDialogApp)
// NOTE - The ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
//////////////////////////////////////////////////////////////////////
///////
//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately
// before the previous line.
#endif //
!defined(AFX_FIRSTDIALOG_H__CDF38DB4_8718_11D0_B02C_0080C81A3AA2
å__INCLUDED_)
```

CFirstDialogApp inherits from CWinApp, which provides most of the functionality. CWinApp has a constructor, which does nothing, as did the SDI and MDI constructors earlier in this chapter, and it overrides the virtual function InitInstance(), as shown in Listing 1.6.

Listing 1.6 FirstDialog.cpp—CDialog16App::InitInstance()

```
BOOL CFirstDialogApp::InitInstance()
{
AfxEnableControlContainer();
// Standard initialization
// If you are not using these features and want to reduce the size
// of your final executable, you should remove from the following
// the specific initialization routines you don't need.
#ifdef _AFXDLL
Enable3dControls(); // Call this when using MFC in a shared DLL
```

```
#else
Enable3dControlsStatic(); // Call this when linking to MFC statically
#endif
CFirstDialogDlg dlg;
m_pMainWnd = &dlg;
int nResponse = dlg.DoModal();
if (nResponse == IDOK)
{
// TODO: Place code here to handle when the dialog is
// dismissed with OK
}
else if (nResponse == IDCANCEL)
{
// TODO: Place code here to handle when the dialog is
// dismissed with Cancel
}
// Because the dialog has been closed, return FALSE so that you exit the
// application, rather than start the application's message pump.
return FALSE;
}
```

This enables 3D controls, because you asked for them, and then puts up the dialog box that is the entire application. To do that, the function declares an instance of CDialog16Dlg, dlg, and then calls the DoModal() function of the dialog, which displays the dialog box onscreen and returns IDOK if the user clicks OK, or IDCANCEL if the user clicks Cancel. (This process is discussed further in Chapter 2.) It's up to you to make that dialog box actually do something. Finally, InitInstance() returns FALSE because this is a dialog-based application and when the dialog box is closed, the application is ended. As you saw earlier for the SDI and MDI applications, InitInstance() usually returns TRUE to mean "everything is fine—run the rest of the application" or FALSE to mean "something went wrong while initializing." Because there is no "rest of the application," dialog-based apps always return FALSE from their InitInstance().

DIALOGS AND CONTROLS

In this chapter

Understanding Dialog Boxes
Creating a Dialog Box Resource
Writing a Dialog Box Class
Using the Dialog Box Class

Understanding Dialog Boxes

Windows programs have a graphical user interface. In the days of DOS, the program could simply print a prompt onscreen and direct the user to enter whatever value the program needed. With Windows, however, getting data from the user is not as simple, and most user input is obtained from dialog boxes. For example, a user can give the application details about a request by typing in edit boxes, choosing from list boxes, selecting radio buttons, checking or unchecking check boxes, and more. These components of a dialog box are called *controls*. Chances are that your Windows application will have several dialog boxes, each designed to retrieve a specific type of information from your user. For each dialog box that appears onscreen, there are two entities you need to develop: a *dialog box resource* and a *dialog box class*. The dialog box resource is used to draw the dialog box and its controls onscreen. The class holds the values of the dialog box, and it is a member function of the class that causes the dialog box to be drawn onscreen. They work together to achieve the overall effect: making communication with the program easier for your user.

You build a dialog box resource with the resource editor, adding controls to it and arranging them to make the control easy to use. Class Wizard then helps you to create a dialog box class, typically derived from the MFC class CDialog, and to connect the resource to the class. Usually, each control on the dialog box resource corresponds to one member variable in the class. To display the dialog box, you call a member function of the class. To set the control values to defaults before displaying the dialog box, or to determine the values of the controls after the user is finished with the box, you use the member variables of the class.

Creating a Dialog Box Resource

The first step in adding a dialog box to your MFC application is creating the dialog box resource, which acts as a sort of template for Windows. When Windows sees the dialog box resource in your program, it uses the commands in the resource to construct the dialog box for you.

In this chapter you learn to work with dialog boxes by adding one to a simple application. Create an SDI application just as you did in Chapter 1, “Building Your First Windows Application,” calling it simply SDI.

You will create a dialog box resource and a dialog box class for the application, write code to display the dialog box, and write code to use the values entered by the user. To create a dialog box resource, first open the application. Choose Insert, Resource from Developer Studio’s menu bar. The Insert Resource dialog box, shown in Figure 2.1, appears. Double-click Dialog in the Resource Type box. The dialog box editor appears, as shown in Figure 2.2.

Bring up the Properties dialog box for the new dialog box by choosing View, Properties. Change the caption to **Sample Dialog**, as shown in Figure 2.3. You'll be using the Properties dialog box quite a lot as you work on this dialog box resource, so pin it to the screen by clicking the pushpin in the upper-left corner.

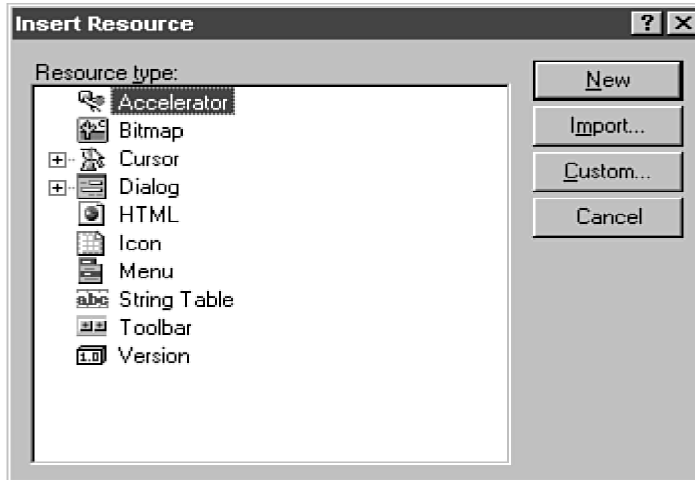


FIG. 2.1 Double-click Dialog on the Insert Resource dialog box.

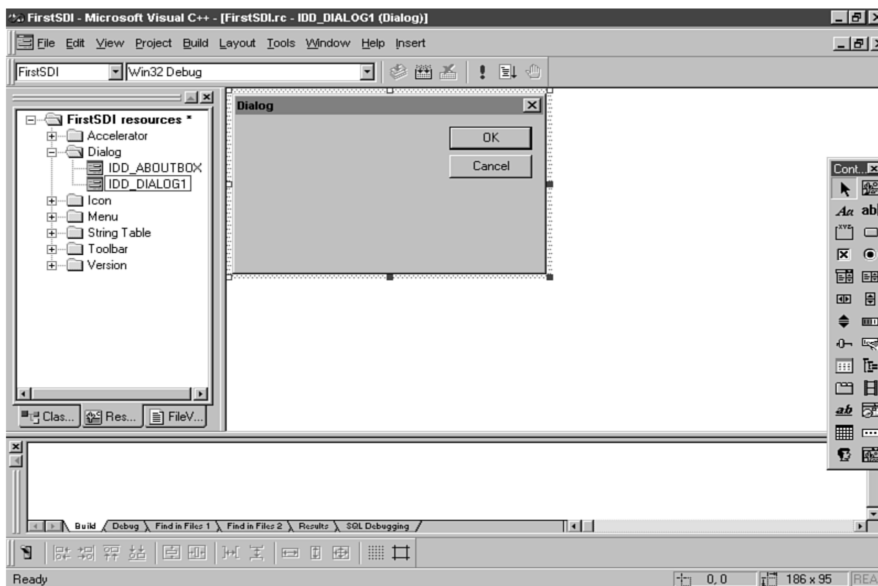


FIG. 2.2 A brand new dialog box resource has a title, an OK button, and a Cancel button.

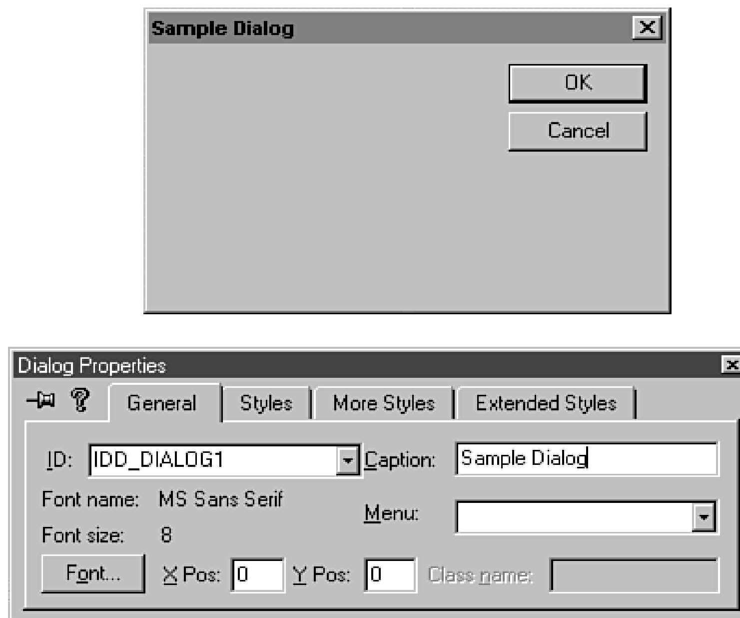


FIG. 2.3 Use the Dialog Properties dialog box to change the title of the new dialog box.

The control palette shown at the far right of Figure 2.2 is used to add controls to the dialog box resource. Dialog boxes are built and changed with a very visual WYSIWYG interface. If you need a button on your dialog box, you grab one from the control palette, drop it where you want it, and change the caption from Button1 to Lookup, or Connect, or whatever you want the button to read. All the familiar Windows controls are available for your dialog boxes:

- *Static text.* Not really a control, this is used to label other controls such as edit boxes.
- *Edit box.* Single line or multiline, this is a place for users to type strings or numbers as input to the program. Read-only edit boxes are used to display text.
- *Button.* Every dialog box starts with OK and Cancel buttons, but you can add as many of your own as you want.
- *Check box.* You use this control to set options on or off; each option can be selected or deselected independently.
- *Radio button.* You use this to select only one of a number of related options. Selecting one button deselects the rest.
- *List box.* You use this box type to select one item from a list hardcoded into the dialog box or filled in by the program as the dialog box is created. The user cannot type in the selection area.
- *Combo box.* A combination of an edit box and a list box, this control enables users to select from a list or type their response, if the one they want isn't on the list.

The sample application in this chapter is going to have a dialog box with a selection of controls on it, to demonstrate the way they are used.

Defining Dialog Box and Control IDs

Because dialog boxes are often unique to an application (with the exception of the common dialog boxes), you almost always create your own IDs for both the dialog box and the controls it contains. You can, if you want, accept the default IDs that the dialog box editor creates for you. However, these IDs are generic (for example, IDD_DIALOG1, IDC_EDIT1, IDC_RADIO1, and so on), so you'll probably want to change them to something more specific. In any case, as you can tell from the default IDs, a dialog box's ID usually begins with the prefix IDD, and control IDs usually begin with the prefix IDC. You change these IDs in the Properties dialog box: Click the control (or the dialog box background to select the entire background), and choose View, Properties unless the Properties dialog box is already pinned in place; then change the resource ID to a descriptive name that starts with IDD for a dialog and IDC for a control.

Creating the Sample Dialog Box

Click the Edit box button on the control palette, and then click in the upper-left corner of the dialog box to place the edit box. If necessary, grab a moving handle and move it until it is in approximately the same place as the edit box in Figure 2.4. Normally, you would change the ID from Edit1, but for this sample leave it unchanged.

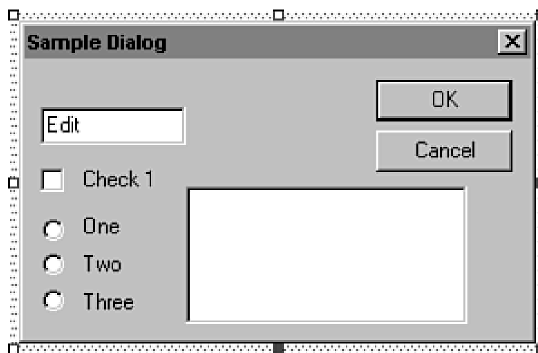


FIG. 2.4 You can build a simple dialog box quickly in the resource editor.

Add a check box and three radio buttons to the dialog box so that it resembles Figure 2.4.

Change the captions on the radio buttons to **One**, **Two**, and **Three**. To align all these controls, click one, and then while holding down the Ctrl key, click each of the rest of them. Choose Layout, Align, Left, and if necessary drag the stack of controls over with the mouse while they are all selected. Then choose Layout, Space Evenly, Down, to adjust the vertical spacing.

Click the One radio button again and bring up the Properties dialog box. Select the Group check box. This indicates that this is the first of a group of buttons. When you select a radio button, all the other buttons in the group are deselected. Add a list box to the dialog box, to the right of the radio buttons, and resize it to match Figure 2.4. With the list box highlighted, choose View, Properties to bring up the Properties dialog box if it is not still pinned in place. Select the Styles tab and make sure that the Sort box is not selected. When this box is selected, the strings in your list box are

automatically presented in alphabetical order. For this application, they should be presented in the order that they are added.

Writing a Dialog Box Class

When the resource is complete, bring up ClassWizard by choosing View, ClassWizard. ClassWizard recognizes that this new dialog box resource does not have a class associated with it and offers to build one for you, as shown in Figure 2.5. Leave the Create a New Class radio button selected, and click OK. The New Class dialog box appears, as shown in Figure 2.6. Fill in the classname as CSdiDialog and click OK. ClassWizard creates a new class, prepares the source file (SdiDialog.cpp) and header file (SdiDialog.h), and adds them to your project.

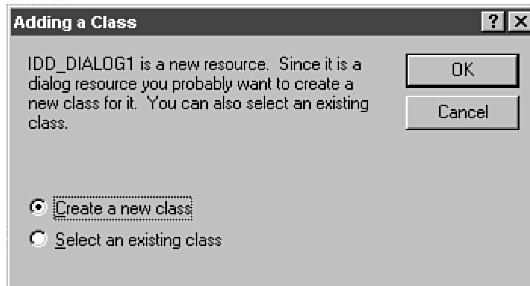


FIG. 2.5 ClassWizard makes sure you don't forget to create a class to go with your new dialog box resource.

You connect the dialog box resources to your code with the Member Variables tab of ClassWizard, shown in Figure 2.7. Click IDC_CHECK1 and then click the Add Variable button.This brings up the Add Member Variable dialog box, shown in Figure 2.8.

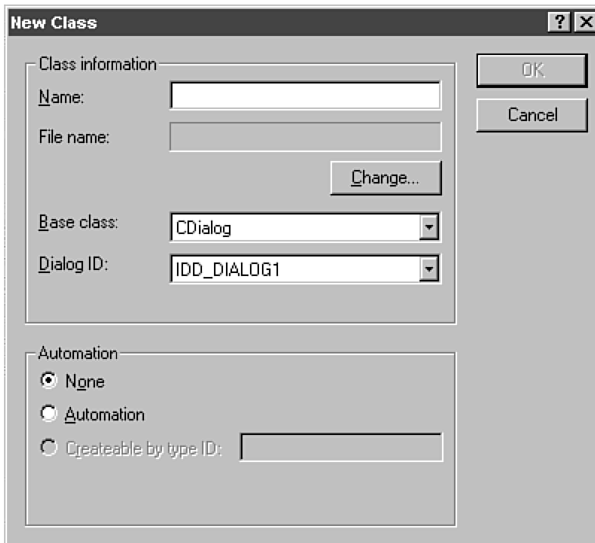


FIG. 2.6 Creating a dialog box class is simple with ClassWizard.

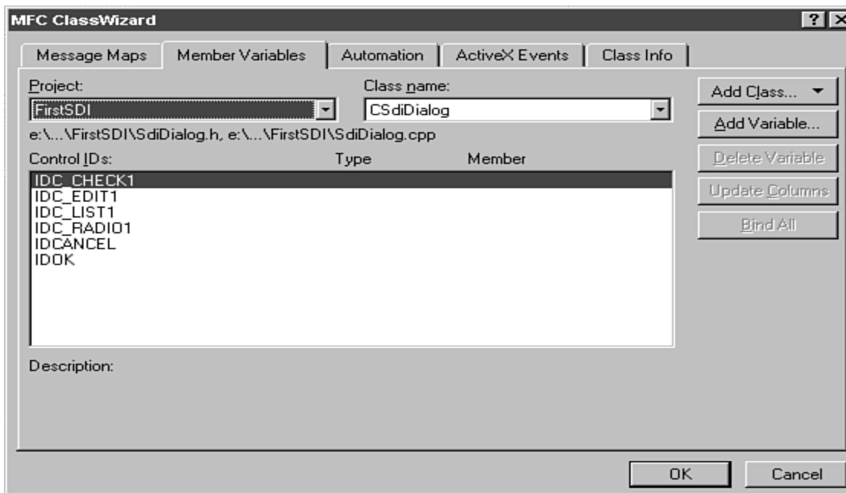


FIG. 2.7 The Member Variables tab of ClassWizard connects dialog box controls to dialog box class member variables.

A member variable in the new dialog box class can be connected to a control's value or to the control. This sample demonstrates both kinds of connection. For IDC_CHECK1, fill in the variable name as m_check, and make sure that the Category drop-down box has Value selected. If you open the Variable Type drop-down box, you will see that the only possible choice is BOOL. Because a check box can be either selected or not selected, it can be connected only to a BOOL variable, which holds the value TRUE or FALSE. Click OK to complete the connection.

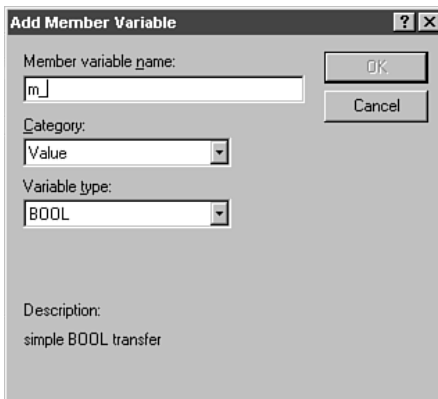


FIG. 2.8 You choose the name for the member variable associated with each control.

Here are the data types that go with each control type:

- *Edit box.* Usually a string but also can be other data types, including int, float, and long
- *Check box.* Int
- *Radio button.* Int
- *List box.* String

- *Combo box.* String
- *Scrollbar.* int

Connect IDC_EDIT1 in the same way, to a member variable called m_edit of type CString as a Value. Connect IDC_LIST1 as a Control to a member variable called m_listbox of type CListBox. Connect IDC_RADIO_1, the first of the group of radio buttons, as a Value to an int member variable called m_radio. After you click OK to add the variable, ClassWizard offers, for some kinds of variables, the capability to validate the user's data entry. For example, when an edit control is selected, a field under the variables list allows you to set the maximum number of characters the user can enter into the edit box (see Figure 2.9). Set it to 10 for m_edit. If the edit box is connected to a number (int or float), this area of ClassWizard is used to specify minimum or maximum values for the number entered by the user. The error messages asking the user to try again are generated automatically by MFC with no work on your part.

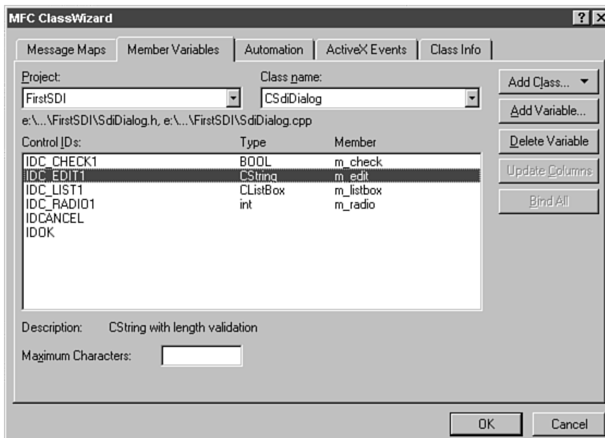


FIG. 2.9 Enter a number in the Maximum Characters field to limit the length of a user's entry.

Using the Dialog Box Class

Now that you have your dialog box resource built and your dialog box class written, you can create objects of that class within your program and display the associated dialog box element. The first step is to decide what will cause the dialog box to display. Typically, it is a menu choice, but because adding menu items and connecting them to code are not covered until Chapter 8, "Building a Complete Application: ShowString," you can simply have the dialog box display when the application starts running. To display the dialog box, you call the DoModal() member function of the dialog box class.

Modeless Dialog Boxes Most of the dialog boxes you will code will be modal dialog boxes. A modal dialog box is on top of all the other windows in the application: The user must deal with the dialog box and then close it before going on to other work. An example of this is the dialog box that comes up when the user chooses File, Open in any Windows application. A modeless dialog box enables the user to click the underlying application and do some other work and then return to the dialog box. An example of this is the dialog box that comes up when the user chooses Edit, Find in

many Windows applications. Displaying a modeless dialog box is more difficult than displaying a modal one. The dialog box object, the instance of the dialog box class, must be managed carefully. Typically, it is created with new and destroyed with delete when the user closes the dialog box with Cancel or OK. You have to override a number of functions within the dialog box class. In short, you should be familiar and comfortable with modal dialog boxes before you attempt to use a modeless dialog box. When you're ready, look at the Visual C++ sample called MODELESS that comes with Developer Studio. The fastest way to open this sample is by searching for MODELESS in InfoViewer. Searching in InfoViewer is covered in Appendix C, "The Visual Studio User Interface, Menus, and Toolbars."

Arranging to Display the Dialog Box

Select the ClassView in the project workspace pane, expand the SDI Classes item, and then expand CSdiApp. Double-click the InitInstance() member function. This function is called whenever the application starts. Scroll to the top of the file, and after the other #include statements, add this directive:

```
#include "sdidialog.h"
```

This ensures that the compiler knows what a CSdiDialog class is when it compiles this file. Double-click InitInstance() in the ClassView again to bring the cursor to the beginning of the function. Scroll down to the end of the function, and just before the return at the end of the function, add the lines in Listing 2.1.

Listing 2.1 SDI.CPP—Lines to Add at the End of CSdiApp::InitInstance()

```
CSdiDialog dlg;
dlg.m_check = TRUE;
dlg.m_edit = "hi there";
CString msg;
if (dlg.DoModal() == IDOK)
{
msg = "You clicked OK. ";
}
else
{
msg = "You cancelled. ";
}
msg += "Edit box is: ";
msg += dlg.m_edit;
AfxMessageBox (msg);
```

Entering Code

As you enter code into this file, you may want to take advantage of a feature that makes its debut in this version of Visual C++: Autocompletion. Covered in more detail in Appendix C, Autocompletion saves you the trouble of remembering all the member variables and functions of a class. If you type dlg. and then pause, a window will appear, listing all the member variables and functions of the class CSdiDialog, including those it inherited from its base class. If you start to type the variable you want—for example, typing m_—the list will scroll to variables starting with m_. Use the

arrow keys to select the one you want, and press Space to select it and continue typing code. You are sure to find this feature a great time saver. If the occasional pause as you type bothers you, Autocompletion can be turned off by choosing Tools, Options and clicking the Editor tab. Deselect the parts of Autocompletion you no longer want.

This code first creates an instance of the dialog box class. It sets the check box and edit box to simple default values. (The list box and radio buttons are a little more complex and are added later in this chapter, in “Using a List Box Control” and “Using Radio Buttons.”) The dialog box displays onscreen by calling its DoModal() function, which returns a number represented by IDOK if the user clicks OK and IDCANCEL if the user clicks Cancel. The code then builds a message and displays it with the AfxMessageBox function.

Build the project by choosing Build, Build or by clicking the Build button on the Build toolbar. Run the application by choosing Build, Execute or by clicking the Execute Program button on the Build toolbar. You will see that the dialog box displays with the default values you just coded, as shown in Figure 2.10. Change them, and click OK. You should get a message box telling you what you did, such as the one in Figure 2.11. Now the program sits there, ready to go, but because there is no more for it to do, you can close it by choosing File, Exit or by clicking the – in the top-right corner.

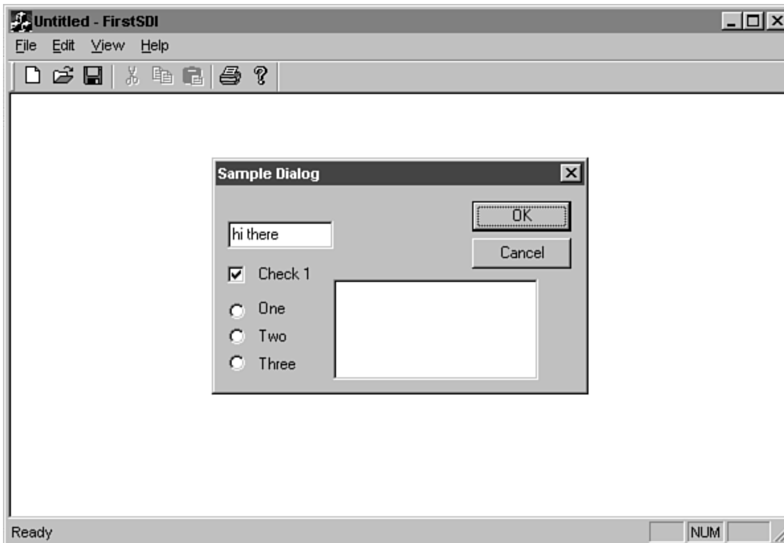


FIG. 2.10 Your application displays the dialog box when it first runs.

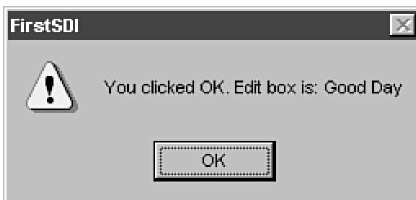


FIG. 2.11 After you click OK, the application echoes the contents of the edit control

Run it again, change the contents of the edit box, and this time click Cancel on the dialog box. Notice in Figure 2.12 that the edit box is reported as still hi there. This is because MFC does not copy the control values into the member variables when the user clicks Cancel. Again, just close the application after the dialog box is gone.

FIG. 2.12 When you click Cancel, the application ignores any changes you made.

Be sure to try entering more characters into the edit box than the 10 you specified with ClassWizard. You will find you cannot type more than 10 characters—the system just beeps at you. If you try to paste in something longer than 10 characters, only the first 10 characters appear in the edit box.

Behind the Scenes

You may be wondering what's going on here. When you click OK on the dialog box, MFC arranges for a function called OnOK() to be called. This function is inherited from CDialog, the base class for CSdiDialog. Among other things, it calls a function called DoDataExchange(), which ClassWizard wrote for you. Here's how it looks at the moment:

```
void CSdiDialog::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CSdiDialog)
DDX_Control(pDX, IDC_LIST1, m_listbox);
DDX_Check(pDX, IDC_CHECK1, m_check);
DDX_Text(pDX, IDC_EDIT1, m_edit);
DDV_MaxChars(pDX, m_edit, 10);
DDX_Radio(pDX, IDC_RADIO1, m_radio);
//}}AFX_DATA_MAP
}
```

The functions with names that start with DDX all perform data exchange: Their second parameter is the resource ID of a control, and the third parameter is a member variable in this class. This is the way that ClassWizard connected the controls to member variables—by generating this code for you. Remember that ClassWizard also added these variables to the dialog box class by generating code in the header file that declares them. There are 34 functions whose names begin with DDX: one for each type of data that might be exchanged between a dialog box and a class. Each has the type in its name. For example, DDX_Check is used to connect a check box to a BOOL member variable. DDX_Text is used to connect an edit box to a CString member variable. ClassWizard chooses the right function name when you make the connection.

Functions with names that start with DDV perform data validation. ClassWizard adds a call to DDV_MaxChars right after the call to DDX_Text that filled m_edit with the contents of IDC_EDIT1. The second parameter of the call is the member variable name, and the third is the limit: how many characters can be in the string. If a user ever managed to get extra characters into a length-validated string, the DDV_MaxChars() function contains code that puts up a warning box and gets the user to try again. You can just set the limit and count on its being enforced.

Using a List Box Control

Dealing with the list box is more difficult because only while the dialog box is onscreen is the list box control a real window. You cannot call a member function of the list box control class unless the dialog box is onscreen. (This is true of any control that you access as a control rather than as a value.) This means that you must initialize the list box (fill it with strings) and use it (determine which string is selected) in functions that are called by MFC while the dialog box is onscreen.

When it is time to initialize the dialog box, just before it displays onscreen, a CDialog function named OnInitDialog() is called. Although the full explanation of what you are about to do will have to wait until Chapter 3, “Messages and Commands,” follow the upcoming steps to add the function to your class.

In ClassView, right-click CSdiDialog and choose Add Windows Message Handler. The New Windows Message and Event Handlers dialog box shown in Figure 2.13 appears. Choose WM_INITDIALOG from the list and click Add Handler. The message name disappears from the left list and appears in the right list. Click it and then click Edit Existing to see the code.

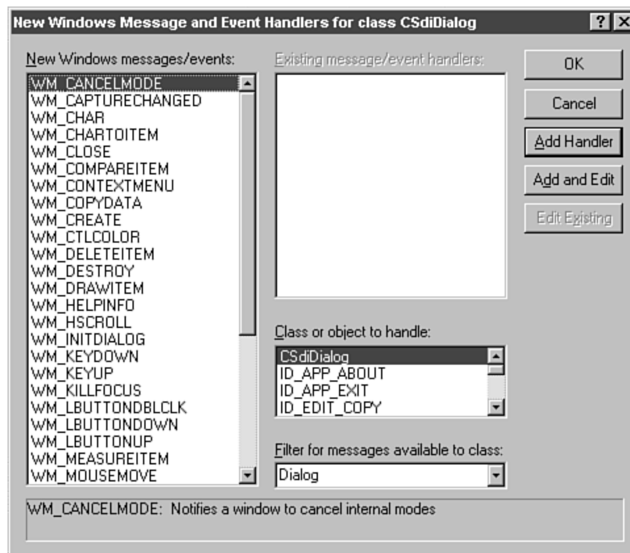


FIG. 2.13 The New Windows Message and Event Handlers dialog box helps you override OnInitDialog().

Remove the TODO comment and add calls to the member functions of the list box so that the function is as shown in Listing 2.2.

Listing 2.2 SDIDIALOG.CPP—CSdiDialog::OnInitDialog()

```
BOOL CSdiDialog::OnInitDialog()
{
CDialog::OnInitDialog();
m_listbox.AddString("First String");
m_listbox.AddString("Second String");
m_listbox.AddString("Yet Another String");
m_listbox.AddString("String Number Four");
m_listbox.SetCurSel(2);
return TRUE; // return TRUE unless you set the focus to a control
// EXCEPTION: OCX Property Pages should return FALSE
}
```

This function starts by calling the base class version of OnInitDialog() to do whatever behind the-scenes work MFC does when dialog boxes are initialized. Then it calls the list box member function AddString() which, as you can probably guess, adds a string to the list box. The strings will be displayed to the user in the order that they were added with AddString(). The final call is to SetCurSel(), which sets the current selection. As you see when you run this program, the index you pass to SetCurSel() is zero based, which means that item 2 is the third in the list, counting 0, 1, 2.

In order to have the message box display some indication of what was selected in the list box, you have to add another member variable to the dialog box class. This member variable will be set as the dialog box closes and can be accessed after it is closed. In ClassView, right-click CSdiDialog and choose Add Member Variable. Fill in the dialog box, as shown in Figure 2.14, and then click OK. This adds the declaration of the CString called m_selected to the header file for you. (If the list box allowed multiple selections, you would have to use a CStringArray to hold the list of selected items.) Strictly speaking, the variable should be private, and you should either add a public accessor function or make CSdiApp::InitInstance() a friend function to CSdiDialog in order to be truly object oriented. Here you take an excusable shortcut. The general rule still holds: Member variables should be private.

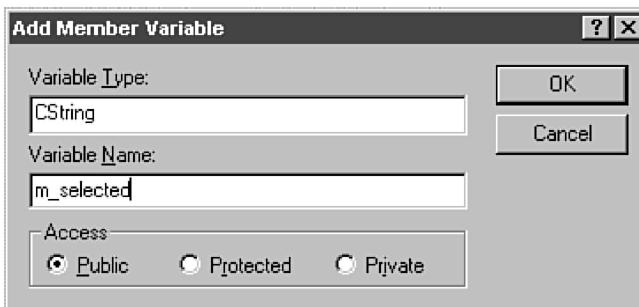


FIG. 2.14 Add a CString to your class to hold the string that was selected in the list box.

This new member variable is used to hold the string that the user selected. It is set when the user clicks OK or Cancel. To add a function that is called when the user clicks OK, follow these steps:

1. Right-click CSdiDialog in the ClassView, and choose Add Windows Message Handler.

2. In the New Windows Message and Event Handlers dialog box, shown in Figure 2.15, highlight ID_OK in the list box at the lower right, labeled Class or Object to Handle.

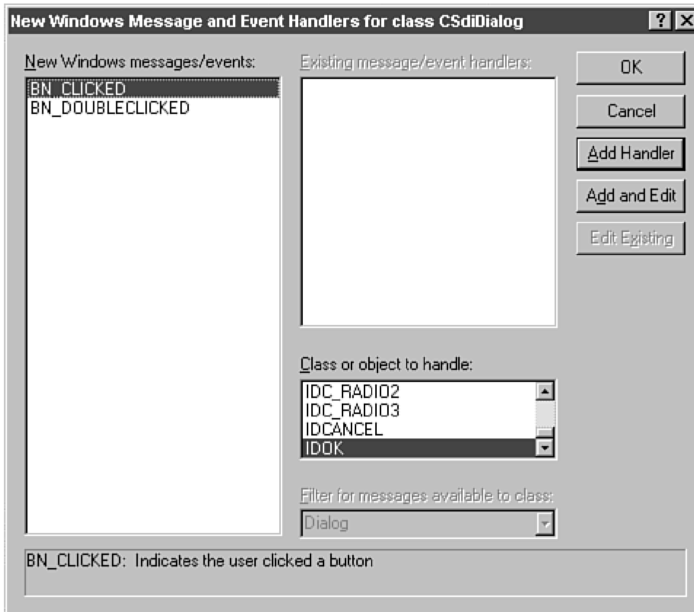


FIG. 2.15 Add a function to handle the user's clicking OK on your dialog box.

3. In the far right list box, select BN_CLICKED. You are adding a function to handle the user's clicking the OK button once.

4. Click the Add Handler button. The Add Member Function dialog box shown in Figure
2.16 appears.

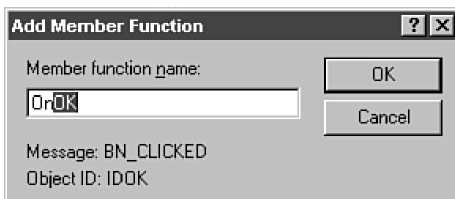


FIG. 2.16 ClassWizard suggests a very good name for this event handler: Do not change it.

5. Accept the suggested name, OnOK(), by clicking OK.

6. Click the Edit Existing button to edit the code, and add lines as shown in Listing 2.3.

Listing 2.3 SDIDIALOG.CPP—CSdiDialog::OnOK()

```
void CSdiDialog::OnOK()
{
int index = m_listbox.GetCurSel();
if (index != LB_ERR)
{
m_listbox.GetText(index, m_selected);
}
else
{
m_selected = “”;
}
CDialog::OnOK();
}
```

This code calls the list box member function GetCurSel(), which returns a constant represented by LB_ERR if there is no selection or if more than one string has been selected. Otherwise, it returns the zero-based index of the selected string. The GetText() member function fills m_selected with the string at position index. After filling this member variable, this function calls the base class OnOK() function to do the other processing required. In a moment you will add lines to CSdiApp::InitInstance() to mention the selected string in the message box. Those lines will execute whether the user clicks OK or Cancel, so you need to add a function to handle the user’s clicking Cancel. Simply follow the numbered steps for adding OnOK, except that you choose ID_CANCEL from the top-right box and agree to call the function OnCancel. The code, as shown in Listing 2.4, resets m_selected because the user canceled the dialog box.

Listing 2.4 SDIDIALOG.CPP—CSdiDialog::OnCancel()

```
void CSdiDialog::OnCancel()
{
m_selected = “”;
CDialog::OnCancel();
}
```

Add these lines to CSdiApp::InitInstance() just before the call to AfxMessageBox():

```
msg += “. List Selection: “;
msg += dlg.m_selected;
```

Build the application, run it, and test it. Does it work as you expect? Does it resemble Figure 2.17?

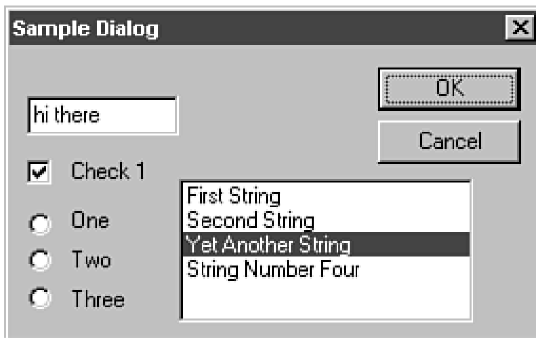


FIG. 2.17 Your application now displays strings in the list box.

Using Radio Buttons

You may have already noticed that when the dialog box first appears onscreen, none of the radio buttons are selected. You can arrange for one of them to be selected by default: Simply add two lines to CSdiDialog::OnInitDialog(). These lines set the second radio button and save the change to the dialog box:

```
m_radio = 1;
UpdateData(FALSE);
```

You may recall that m_radio is the member variable to which the group of radio buttons is connected. It is a zero-based index into the group of buttons, indicating which one is selected. Button 1 is the second button. The call to UpdateData() refreshes the dialog box controls with the member variable values. The parameter indicates the direction of transfer: UpdateData(TRUE) would refresh the member variables with the control values, wiping out the setting of m_radio you just made.

Unlike list boxes, a group of radio buttons can be accessed after the dialog box is no longer onscreen, so you won't need to add code to OnOK() or OnCancel(). However, you have a problem: how to convert the integer selection into a string to tack on the end of msg. There are lots of approaches, including the Format() function of CString, but in this case, because there are not many possible selections, a switch statement is readable and quick. At the end of CSdiApp::InitInstance(), add the lines in Listing 2.5 just before the call to AfxMessageBox().

Listing 2.5 SDIDIALOG.CPP—Lines to Add to CSdiApp::InitInstance()

```
msg += "\r\n";
msg += "Radio Selection: ";
switch (dlg.m_radio)
{
case 0:
msg += "0";
break;
case 1:
msg += "1";
```

```
break;
case 2:
msg += "2";
break;
default:
msg += "none";
break;
}
```

The first new line adds two special characters to the message. *Return*, represented by \r, and *new line*, represented by \n, combine to form the Windows end-of-line marker. This adds a line break after the part of the message you have built so far. The rest of msg will appear on the second line of the message box. The switch statement is an ordinary piece of C++ code, which was also present in C. It executes one of the case statements, depending on the value of dlg.m_radio. Once again, build and test the application. Any surprises? It should resemble Figure 2.18. You are going to be building and using dialog boxes throughout this book, so take the time to understand how this application works and what it does. You may want to step through it with the debugger and watch it in action. You can read all about debugging in the coming Chapter, "Improving Your Application's Performance," and in Appendix D, "Debugging."

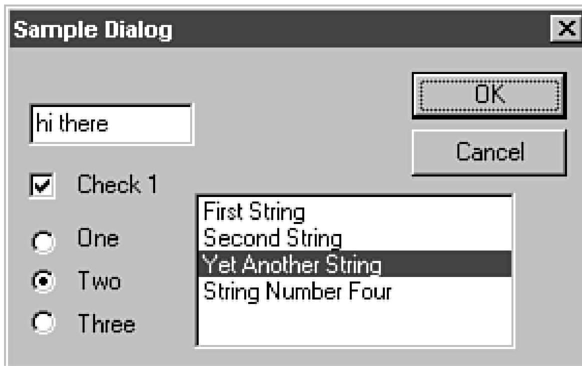


FIG. 2.18 Your application now selects Button Two by default.

MESSAGES AND COMMANDS

In this chapter

Understanding Message Routing

Understanding Message Loops

Reading Message Maps

Learning How ClassWizard Helps You Catch Messages

Recognizing Messages

Understanding Commands

Understanding Command Updates

Learning How ClassWizard Helps You Catch Commands and Command Updates

Understanding Message Routing

If there is one thing that sets Windows programming apart from other kinds of programming, it is messages. Most DOS programs, for example, relied on watching (sometimes called *polling*) possible sources of input like the keyboard or the mouse to await input from them. A program that wasn't polling the mouse would not react to mouse input. In contrast, everything that happens in a Windows program is mediated by messages. A message is a way for the operating system to tell an application that something has happened—for example, the user has typed, clicked, or moved the mouse, or the printer has become available. A window (and every screen element is a window) can also send a message to another window, and typically most windows react to messages by passing a slightly different message along to another window. MFC has made it much easier to deal with messages, but you must understand what is going on beneath the surface.

Messages are all referred to by their names, though the operating system uses integers to refer to them. An enormous list of #define statements connects names to numbers and lets Windows programmers talk about WM_PAINT or WM_SIZE or whatever message they need to talk about. (The WM stands for Window Message.) An excerpt from that list is shown in Listing 3.1.

Listing 3.1 Excerpt from winuser.h Defining Message Names

```
#define WM_SETFOCUS          0x0007
#define WM_KILLFOCUS         0x0008
#define WM_ENABLE            0x000A
#define WM_SETREDRAW         0x000B
#define WM_SETTEXT           0x000C
#define WM_GETTEXT           0x000D
#define WM_GETTEXTLENGTH     0x000E
#define WM_PAINT             0x000F
#define WM_CLOSE             0x0010
```

```
#define WM_QUERYENDSESSION      0x0011
#define WM_QUIT                 0x0012
#define WM_QUERYOPEN            0x0013
#define WM_ERASEBKGND           0x0014
#define WM_SYSCOLORCHANGE       0x0015
#define WM_ENDSESSION           0x0016
```

As well as a name, a message knows what window it is for and can have up to two parameters. (Often, several different values are packed into these parameters, but that's another story.) Different messages are handled by different parts of the operating system or your application. For example, when the user moves the mouse over a window, the window receives a WM_MOUSEMOVE message, which it almost certainly passes to the operating system to deal with. The operating system redraws the mouse cursor at the new location. When the left button is clicked over a button, the button (which is a window) receives a WM_LBUTTONDOWN message and handles it, often generating another message to the window that contains the button, saying, in effect, "I was clicked." MFC has enabled many programmers to completely ignore low-level messages such as WM_MOUSEMOVE and WM_LBUTTONDOWN. Instead, programmers deal only with higher level messages that mean things like "The third item in this list box has been selected" or "The Submit button has been clicked." All these kinds of messages move around in your code and the operating system code in the same way as the lower level messages. The only difference is what piece of code chooses to handle them. MFC makes it much simpler to announce, at the individual class's level, which messages each class can handle. The old C way, which you will see in the next section, made those announcements at a higher level and interfered with the object-oriented approach to Windows programming, which involves hiding implementation details as much as possible inside objects.

Understanding Message Loops

The heart of any Windows program is the message loop, typically contained in a WinMain() routine. The WinMain() routine is, like the main() in DOS or UNIX, the function called by the operating system when you run the program. You won't write any WinMain() routines because it is now hidden away in the code that AppWizard generates for you. Still, there is a WinMain(), just as there is in Windows C programs. Listing 3.2 shows a typical WinMain().

Listing 3.2 Typical WinMain() Routine

```
int APIENTRY WinMain(HINSTANCE hInstance,
HINSTANCE hPrevInstance,
LPSTR lpCmdLine,
int nCmdShow)
{
MSG msg;
if (! InitApplication (hInstance))
return (FALSE);
if (! InitInstance (hInstance, nCmdShow))
return (FALSE);
```

```
while (GetMessage (&msg, NULL, 0, 0)){
TranslateMessage (&msg);
DispatchMessage (&msg);
}
return (msg.wParam);
}
```

In a Windows C program like this, InitApplication() typically calls RegisterWindow(), and InitInstance() typically calls CreateWindow(). (More details on this are in Appendix B, "Windows Programming Review and a Look Inside Cwnd.") Then comes the message loop, the while loop that calls GetMessage(). The API function GetMessage() fills msg with a message destined for this application and almost always returns TRUE, so this loop runs over and over until the program is finished. The only thing that makes GetMessage() return FALSE is if the message it receives is WM_QUIT.

TranslateMessage() is an API function that streamlines dealing with keyboard messages. Most of the time, you don't need to know that "the A key just went down" or "the A key just went up," and so on. It's enough to know that "the user pressed A." TranslateMessage() deals with that. It catches the WM_KEYDOWN and WM_KEYUP messages and usually sends a WM_CHAR message in their place. Of course, with MFC, most of the time you don't care that the user pressed A. The user types into an edit box or similar control, and you can retrieve the entire string out of it later, when the user has clicked OK. Don't worry too much about TranslateMessage().

The API function DispatchMessage() calls the WndProc for the window that the message is headed for. The WndProc() function for a Windows C program is a huge switch statement with one case for each message the programmer planned to catch, such as the one in

Listing 3.3.

Listing 3.3 Typical WndProc() Routine

```
LONG APIENTRY MainWndProc (HWND hWnd, // window handle
UINT message, // type of message
UINT wParam, // additional information
LONG lParam) // additional information
{
switch (message) {
case WM_MOUSEMOVE:
//handle mouse movement
break;
case WM_LBUTTONDOWN:
//handle left click
break;
case WM_RBUTTONDOWN:
//handle right click
break;
```

```
case WM_PAINT:
//repaint the window
break;
case WM_DESTROY: // message: window being destroyed
PostQuitMessage (0);
break;
default:
return (DefWindowProc (hWnd, message, wParam, lParam));
}
return (0);
}
```

As you can imagine, these WndProcs become very long in a hurry. Program maintenance can be a nightmare. MFC solves this problem by keeping information about message processing close to the functions that handle the messages, freeing you from maintaining a giant switch statement that is all in one place. Read on to see how it's done.

Reading Message Maps

Message maps are part of the MFC approach to Windows programming. Instead of writing a WinMain() function that sends messages to your WindProc and then writing a WindProc that checks which kind of message this is and then calls another of your functions, you just write the function that will handle the message, and you add a message map to your class that says, in effect, "I will handle this sort of message." The framework handles whatever routing is required to send that message to you.

Message maps come in two parts: one in the .h file for a class and one in the corresponding .cpp. Typically, they are generated by wizards, although in some circumstances you will add entries yourself. Listing 3.4 shows the message map from the header file of one of the classes in a simple application called ShowString, presented in Chapter 8, "Building a Complete Application: ShowString."

Listing 3.4 Message Map from showstring.h

```
//{{AFX_MSG(CShowStringApp)
afx_msg void OnAppAbout();
// NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
```

This declares a function called OnAppAbout(). The specially formatted comments around the declarations help ClassWizard keep track of which messages are caught by each class.

DECLARE_MESSAGE_MAP() is a macro, expanded by the C++ compiler's preprocessor, that declares some variables and functions to set up some of this magic message catching. The message map in the source file, as shown in Listing 3.5, is quite similar.

Listing 3.5 Message Map from Chapter 8's showstring.cpp

```
BEGIN_MESSAGE_MAP(CShowStringApp, CWinApp)
//{{AFX_MSG_MAP(CShowStringApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
// NOTE - the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
// Standard file based document commands
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
// Standard print setup command
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

Message Map Macros

BEGIN_MESSAGE_MAP and END_MESSAGE_MAP are macros that, like DECLARE_MESSAGE_MAP in the include file, declare some member variables and functions that the framework can use to navigate the maps of all the objects in the system. A number of macros are used in message maps, including these:

- DECLARE_MESSAGE_MAP—Used in the include file to declare that there will be a message map in the source file.
- BEGIN MESSAGE MAP—Marks the beginning of a message map in the source file.
- END MESSAGE MAP—Marks the end of a message map in the source file.
- ON_COMMAND—Used to delegate the handling of a specific command to a member function of the class.
- ON_COMMAND_RANGE—Used to delegate the handling of a group of commands, expressed as a range of command IDs, to a single member function of the class.
- ON_CONTROL—Used to delegate the handling of a specific custom control–notification message to a member function of the class.
- ON_CONTROL_RANGE—Used to delegate the handling of a group of custom control–notification messages, expressed as a range of control IDs, to a single member function of the class.
- ON_MESSAGE—Used to delegate the handling of a user-defined message to a member function of the class.
- ON_REGISTERED_MESSAGE—Used to delegate the handling of a registered user-defined message to a member function of the class.
- ON_UPDATE_COMMAND_UI—Used to delegate the updating for a specific command to a member function of the class.
- ON_COMMAND_UPDATE_UI_RANGE—Used to delegate the updating for a group of commands, expressed as a range of command IDs, to a single member function of the class.

- ON_NOTIFY—Used to delegate the handling of a specific control-notification message with extra data to a member function of the class.
- ON_NOTIFY_RANGE—Used to delegate the handling of a group of control-notification messages with extra data, expressed as a range of child identifiers, to a single member function of the class. The controls that send these notifications are child windows of the window that catches them.
- ON_NOTIFY_EX—Used to delegate the handling of a specific control-notification message with extra data to a member function of the class that returns TRUE or FALSE to indicate whether the notification should be passed on to another object for further reaction.
- ON_NOTIFY_EX_RANGE—Used to delegate the handling of a group of control-notification messages with extra data, expressed as a range of child identifiers, to a single member function of the class that returns TRUE or FALSE to indicate whether the notification should be passed on to another object for further reaction. The controls that send these notifications are child windows of the window that catches them.

In addition to these, there are about 100 macros, one for each of the more common messages, that direct a single specific message to a member function. For example, ON_CREATE delegates the WM_CREATE message to a function called OnCreate(). You cannot change the function names in these macros. Typically, these macros are added to your message map by ClassWizard, as demonstrated in Chapter 8.

How Message Maps Work

The message maps presented in Listings 3.3 and 3.4 are for the CShowStringApp class of the ShowString application. This class handles application-level tasks such as opening a new file or displaying the About box. The entry added to the header file's message map can be read as "there is a function called OnAppAbout() that takes no parameters." The entry in the source file's map means "when an ID_APP_ABOUT command message arrives, call OnAppAbout()." It shouldn't be a big surprise that the OnAppAbout() member function displays the About box for the application.

If you don't mind thinking of all this as magic, it might be enough to know that adding the message map entry causes your code to run when the message is sent. Perhaps you're wondering just how message maps really work. Here's how. Every application has an object that inherits from CWinApp, and a member function called Run(). That function calls CWinThread::Run(), which is far longer than the simple WinMain() presented earlier but has the same message loop at its heart: call GetMessage(), call TranslateMessage(), call DispatchMessage(). Almost every window object uses the same old-style Windows class and the same WindProc, called AfxWndProc(). The WindProc, as you've already seen, knows the handle, hWnd, of the window the message is for. MFC keeps something called a *handle map*, a table of window handles and pointers to objects, and the framework uses this to send a pointer to the C++ object, a CWnd*. Next, it calls WindowProc(), a virtual function of that object. Buttons or views might have different WindowProc() implementations, but through the magic of polymorphism, the right function is called.

Polymorphism

Virtual functions and polymorphism are important C++ concepts for anyone working with MFC. They arise only when you are using pointers to objects and when the class of objects to which the pointers are pointing is derived from another class. Consider as an example a class called CDerived that is derived from a base class called CBase, with a member function called Function() that is declared in the base class and overridden in the derived class. There are now two functions: One has the full name CBase::Function(), and the other is CDerived::Function(). If your code has a pointer to a base object and sets that pointer equal to the address of the derived object, it can then call the function, like this:

```
CDerived derivedobject;
CBase* basepointer;
basepointer = &derivedobject;
basepointer->Function();
```

In this case, CBase::Function() will be called. However, there are times when that is not what you want—when you have to use a CBase pointer, but you really want CDerived::Function() to be called. To indicate this, in CBase, Function() is declared to be virtual. Think of it as an instruction to the compiler to override this function, if there is any way to do it. When Function() is declared to be virtual in the base class, CBase, the code fragment above would actually call CDerived::Function(), as desired. That's polymorphism, and that shows up again and again when using MFC classes. You use a pointer to a window, a CWnd*, that really points to a CButton or a CView or some other class derived from CWnd, and when a function such as WindowProc() is called, it will be the derived function—CButton::WindowProc() for example— that is called.

WindowProc()calls OnWndMsg(), the C++ function that really handles messages. First, it checks to see whether this is a message, a command, or a notification. Assuming it's a message, it looks in the message map for the class, using the member variables and functions set up by DECLARE_MESSAGE_MAP, BEGIN_MESSAGE_MAP, and END_MESSAGE_MAP. Part of what those macros arrange is to enable access to the message map entries of the base class by the functions that search the message map of the derived class. That means that if a class inherits from CView and doesn't catch a message normally caught by CView, that message will still be caught by the same CView function as inherited by the derived class. This message map inheritance parallels the C++ inheritance but is independent of it and saves a lot of trouble carrying virtual functions around.

The bottom line: You add a message map entry, and when a message arrives, the functions called by the hidden message loop look in these tables to decide which of your objects, and which member function of the object, should handle the message. That's what's really going on behind the scenes.

Messages Caught by MFC Code

The other great advantage of MFC is that the classes already catch most of the common messages and do the right thing, without any coding on your part at all. For example, you don't need to catch the message that tells you that the user has chosen

File, Save As—MFC classes catch it, put up the dialog box to obtain the new filename, handle all the behind-the-scenes work, and finally call one of your functions, which must be named Serialize(), to actually write out the document. (Chapter 7, "Persistence and File I/O," explains the Serialize() function.) You need only to add message map entries for behavior that is not common to all applications.

Learning How ClassWizard Helps You Catch Messages

Message maps may not be simple to read, but they are simple to create if you use ClassWizard. There are two ways to add an entry to a message map in Visual C++ 6.0: with the main ClassWizard dialog box or with one of the new dialog boxes that add message handlers or virtual functions. This section shows you these dialog boxes for ShowString, rather than work you through creating a sample application.

The ClassWizard Tabbed Dialog Box

The main ClassWizard dialog box is displayed by choosing View, ClassWizard or by pressing Ctrl+W. ClassWizard is a tabbed dialog box, and Figure 3.1 shows the Message Maps tab. At the top of the dialog box are two drop-down list boxes, one that reminds you which project you are working on (ShowString in this case) and the other that reminds you which class owns the message map you are editing. In this case, it is the CShowStringApp class, whose message map you have already seen.

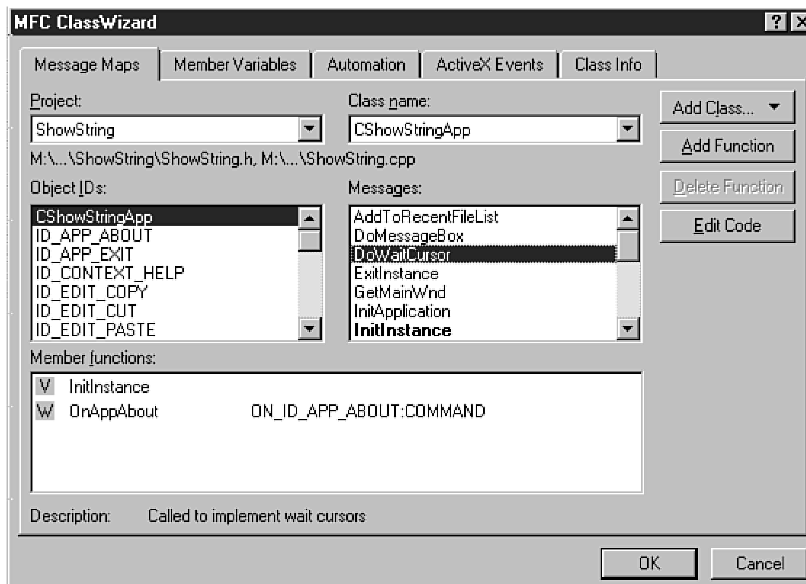


FIG. 3.1 ClassWizard makes catching messages simple.

Below those single-line boxes is a pair of multiline boxes. The one on the left lists the class itself and all the commands that the user interface can generate. Commands are discussed in the "Commands" section later in this chapter. With the classname highlighted, the box on the right lists all the Windows messages this class might catch. It also lists a number of virtual functions that catch common messages.

To the right of those boxes are buttons where you can add a new class to the project, add a function to the class to catch the highlighted message, remove a function that was catching a message, or open the source code for the function that catches the highlighted message. Typically, you select a class, select a message, and click Add Function to catch the message. Here's what the Add Function button sets in motion:

- Adds a skeleton function to the bottom of the source file for the application
- Adds an entry to the message map in the source file
- Adds an entry to the message map in the include file
- Updates the list of messages and member functions in the dialog box

After you add a function, clicking Edit Code makes it simple to start filling in the behavior of that function. If you prefer, double-click the function name in the Member Functions list box. Below the Object IDs and Messages boxes is a list of the member functions of this class that are related to messages. This class has two such functions:

- InitInstance()—Overrides a virtual function in CWinApp, the base class for CShowStringApp, and is labeled with a V (for *virtual* function) in the list.
- OnAppAbout()—Catches the ID_APP_ABOUT command and is labeled with a W (for Windows message) in the list.

The InitInstance function is called whenever an application first starts. You don't need to

understand this function to see that ClassWizard reminds you the function has been overridden.

Finally, under the Member Functions box is a reminder of the meaning of the highlighted

message. called to implement wait cursors is a description of the DoWaitCursor virtual function.

The Add Windows Message Handler Dialog Box

In release 5.0 of Visual C++, a new way of catching messages was added. Rather than opening ClassWizard and then remembering to set the right classname in a drop-down list box, you right-click on the classname in ClassView and then choose Add Windows Message Handler from the shortcut menu that appears. Figure 3.2 shows the dialog box that appears when you make this choice.

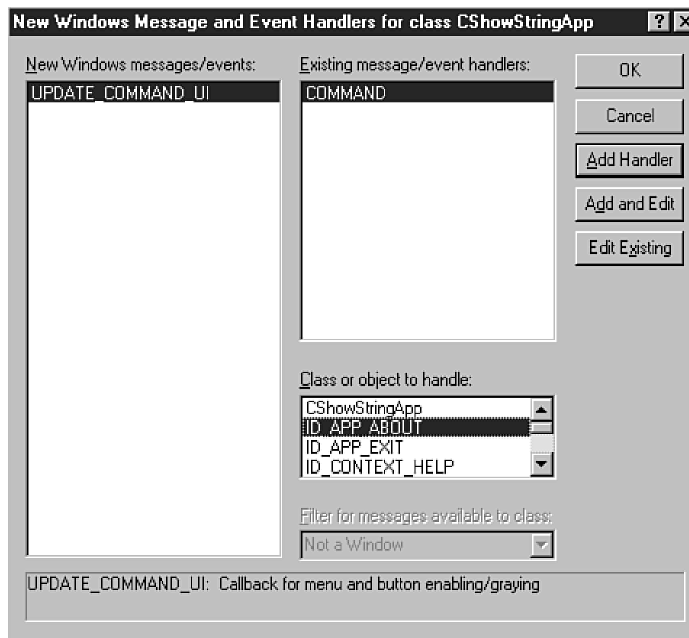


FIG. 3.2 The New Windows Message and Event Handlers dialog box is another way to catch messages.

This dialog box doesn't show any virtual functions that were listed in the main ClassView dia- Ch log box. It is easy to see that this class catches the command ID_APP_ABOUT but doesn't catch the command update. (Commands and command updating are discussed in more detail later in this chapter.) To add a new virtual function, you right-click on the class in ClassView and choose Add New Virtual Function from the shortcut menu. Figure 3.3 shows this dialog box.

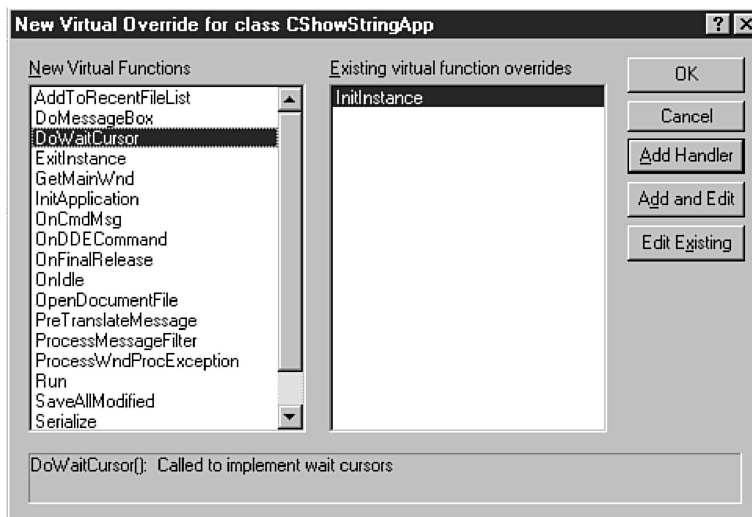


FIG. 3.3 The New Virtual Override dialog box simplifies implementing virtual functions.

You can see in Figure 3.3 that CShowStringApp already overrides the InitInstance() virtual function, and you can see what other functions are available to be overridden. As in the tabbed dialog box, a message area at the bottom of the dialog box reminds you of the purpose of each function: In fact, the text—Called to implement wait cursors—is identical to that in Figure 3.1.

Which Class Should Catch the Message?

The only tricky part of message maps and message handling is deciding which class should catch the message. That's a decision you can't make until you understand all the different message and command targets that make up a typical application. The choice is usually one of the following:

- The active view
- The document associated with the active view
- The frame window that holds the active view
- The application object

Views, documents, and frames are discussed in Chapter 4, "Documents and Views."

Recognizing Messages

There are almost 900 Windows messages, so you won't find a list of them all in this chapter. Usually, you arrange to catch messages with ClassWizard and are presented with a much shorter list that is appropriate for the class you are catching messages with. Not every kind of window can receive every kind of message. For example, only classes that inherit from CListBox receive list box messages such as LB_SETSEL, which directs the list box to move the highlight to a specific list item. The first component of a message name indicates the kind of window this message is destined for, or coming from. These window types are listed in Table 3.1.

Table 3.1 Windows Message Prefixes and Window Types

Prefix	Window Type
ABM, ABN	Appbar
ACM, ACN	Animation control
BM, BN	Button
CB, CBN	Combo box
CDM, CDN	Common dialog box
CPL	Control Panel application
DBT	Any application (device change message)
DL	Drag list box
DM	Dialog box
EM,EN	Edit box
FM,FMEVENT	File Manager
HDM,HDN	Header control
HKM	HotKey control
IMC,IMN	IME window
LB,LBN	List box
LVM,LVN	List view

Prefix	**Window Type**
NM	Any parent window (notification message)
PBM	Progress bar
PBT	Any application (battery power broadcast)
PSM,PSN	Property sheet
SB	Status bar
SBM	Scrollbar
STM,STN	Static control
TB,TBN	Toolbar
TBM	Track bar
TCM,TCN	Tab control
TTM,TTN	ToolTip
TVM,TVN	Tree view
UDM	Up Down control
WM	Generic window

What's the difference between, say, a BM message and a BN message? A BM message is a message to a button, such as "act as though you were just clicked." A BN message is a notification from a button to the window that owns it, such as "I was clicked." The same pattern holds for all the prefixes that end with M or N in the preceding table.

Sometimes the message prefix does not end with M; for example CB is the prefix for a message to a combo box, whereas CBN is the prefix for a notification from a combo box to the window that owns it. Another example is CB_SETCURSEL, a message to a combo box directing it to select one of its strings, whereas CBN_SELCHANGE is a message sent from a combo box, notifying its parent that the user has changed which string is selected.

Understanding Commands

What is a command? It is a special type of message. Windows generates a command whenever a user chooses a menu item, clicks a button, or otherwise tells the system to do something. In older versions of Windows, both menu choices and button clicks generated a WM_COMMAND message; these days you receive a WM_COMMAND for a menu choice and a WM_NOTIFY for a control notification such as button clicking or list box selecting. Commands and notifications are passed around by the operating system just like any other message, until they get into the top of OnWndMsg(). At that point, Windows message passing stops and MFC command routing starts.

Command messages all have, as their first parameter, the resource ID of the menu item that was chosen or the button that was clicked. These resource IDs are assigned according to a standard pattern—for example, the menu item File, Save has the resource ID ID_FILE_SAVE. Command routing is the mechanism OnWndMsg() uses to send the command (or notification) to objects that can't receive messages. Only objects that inherit from CWnd can receive messages, but all objects that inherit from CCmdTarget, including CWnd and CDocument, can receive commands and

notifications. That means a class that inherits from CDocument can have a message map. There won't be any entries in it for messages, only for commands and notifications, but it's still a message map.

How do the commands and notifications get to the class, though? By command routing. (This becomes messy, so if you don't want the inner details, skip this paragraph and the next.) OnWndMsg() calls CWnd::OnCommand() or CWnd::OnNotify(). OnCommand() checks all sorts of petty stuff (such as whether this menu item was grayed after the user selected it but before this piece of code started to execute) and then calls OnCmdMsg(). OnNotify() checks different conditions and then it, too, calls OnCmdMsg(). OnCmdMsg() is virtual, which means that different command targets have different implementations. The implementation for a frame window sends the command to the views and documents it contains.

This is how something that started out as a message can end up being handled by a member function of an object that isn't a window and therefore can't really catch messages.

Should you care about this? Even if you don't care how it all happens, you should care that you can arrange for the right class to handle whatever happens within your application. If the user resizes the window, a WM_SIZE message is sent, and you may have to rescale an image or do some other work inside your view. If the user chooses a menu item, a command is generated, and that means your document can handle it if that's more appropriate. You see examples of these decisions at work in Chapter 4.

Understanding Command Updates

This under-the-hood tour of how MFC connects user actions such as window resizing or menu choices to your code is almost finished. All that's left is to handle the graying of menus and buttons, a process called *command updating*. Imagine you are designing an operating system, and you know it's a good idea to have some menu items grayed to show they can't be used right now. There are two ways you can go about implementing this.

One is to have a huge table with one entry for every menu item and a flag to indicate whether it's available. Whenever you have to display the menu, you can quickly check the table. Whenever the program does anything that makes the item available or unavailable, it updates the table. This is called the *continuous-update approach*.

The other way is not to have a table but to check all the conditions just before your program displays the menu. This is called the *update-on-demand approach* and is the approach taken in Windows. In the old C way of doing things—to check whether each menu option should be grayed—the system sent a WM_INITMENUPOPUP message, which means "I'm about to display a menu." The giant switch in the WindProc caught that message and quickly enabled or disabled each menu item. This wasn't very object-oriented though. In an object-oriented program, different pieces of information are stored in different objects and aren't generally made available to the entire program.

When it comes to updating menus, different objects know whether each item should be grayed. For example, the document knows whether it has been modified since it was last saved, so it can decide whether File, Save should be grayed. However, only the view knows whether some text is currently highlighted; therefore, it can decide if Edit, Cut and Edit, Copy should be grayed. This means that the job of

updating these menus should be parcelled out to various objects within the application rather than handled within the WindProc.

The MFC approach is to use a little object called a CCmdUI, a command user interface, and give this object to whoever catches a CN_UPDATE_COMMAND_UI message. You catch those messages by adding (or getting ClassWizard to add) an ON_UPDATE_COMMAND_UI macro in your message map. If you want to know what's going on behind the scenes, it's this: The operating system still sends WM_INITMENUPOPUP; then the MFC base classes such as CFrameWnd take over. They make a CCmdUI, set its member variables to correspond to the first menu item, and call one of that object's own member functions, DoUpdate(). Then, DoUpdate() sends out the CN_COMMAND_UPDATE_UI message with a pointer to itself as the CCmdUI object the handlers use. The same CCmdUI object is then reset to correspond to the second menu item, and so on, until the entire menu is ready to be displayed. The CCmdUI object is also used to gray and ungray buttons and other controls in a slightly different context.

CCmdUI has the following member functions:

- Enable()—Takes a TRUE or FALSE (defaults to TRUE). This grays the user interface item if FALSE and makes it available if TRUE.
- SetCheck()—Checks or unchecks the item.
- SetRadio()—Checks or unchecks the item as part of a group of radio buttons, only one of which can be set at any time.
- SetText()—Sets the menu text or button text, if this is a button.
- DoUpdate()—Generates the message.

Determining which member function you want to use is usually clear-cut. Here is a shortened version of the message map from an object called CWhoisView, a class derived from CFormView that is showing information to a user. This form view contains several edit boxes, and the user may want to paste text into one of them. The message map contains an entry to catch the update for the ID_EDIT_PASTE command, like this:

```
BEGIN_MESSAGE_MAP(CWhoisView, CFormView)
...
ON_UPDATE_COMMAND_UI(ID_EDIT_PASTE, OnUpdateEditPaste)
...
END_MESSAGE_MAP()
```

The function that catches the update, OnUpdateEditPaste(), looks like this:

```
void CWhoisView::OnUpdateEditPaste(CCmdUI* pCmdUI)
{
pCmdUI->Enable(::IsClipboardFormatAvailable(CF_TEXT));
}
```

This calls the API function ::IsClipboardFormatAvailable() to see whether there is text in the Clipboard. Other applications may be able to paste in images or other nontext Clipboard contents, but this application cannot and grays the menu item if there is no text available to paste. Most command update functions look just like this: They call Enable() with a parameter that is a call to a function that returns TRUE or FALSE, or perhaps a simple logical expression. Command update handlers must be fast because five to ten of them must run between the moment the user clicks to display the menu and the moment before the menu is actually displayed.

Learning How ClassWizard Helps You Catch Commands and Command Updates

The ClassWizard dialog box shown in Figure 4.1 has the classname highlighted in the box labeled Object IDs. Below that are resource IDs of every resource (menu, toolbar, dialog box controls, and so on) that can generate a command or message when this object (view, dialog, and so on) is on the screen. If you highlight one of those, the list of messages associated with it is much smaller, as you see in Figure 3.4.

Only two messages are associated with each resource ID: COMMAND and UPDATE_COMMAND_UI. The first enables you to add a function to handle the user selecting the menu option or clicking the button—that is, to catch the command. The second enables you to add a function to set the state of the menu item, button, or other control just as the operating system is about to display it—that is, to update the command. (The COMMAND choice is boldface in Figure 3.4 because this class already catches that command.)

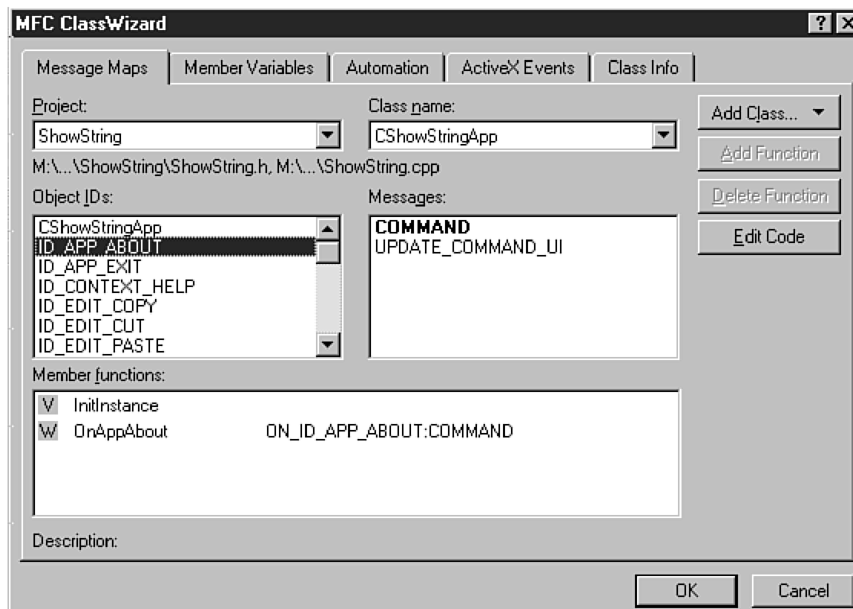


FIG. 3.4 ClassWizard enables you to catch or update commands.

Clicking Add Function to add a function that catches or updates a command involves an extra step. ClassWizard gives you a chance to change the default function name, as shown in Figure 3.5. This is almost never appropriate. There is a regular pattern to the suggested names, and experienced MFC programmers come to count on function names that follow that pattern. Command handler functions, like message handlers, have names that start with On. Typically, the remainder of the function name is formed by removing the ID and the underscores from the resource ID and capitalizing each word. Command update handlers have names that start with OnUpdate and use the same conventions for the remainder of the function name. For example, the function that catches ID_APP_EXIT should be called OnAppExit(), and the function that updates ID_APP_EXIT should be called OnUpdateAppExit().

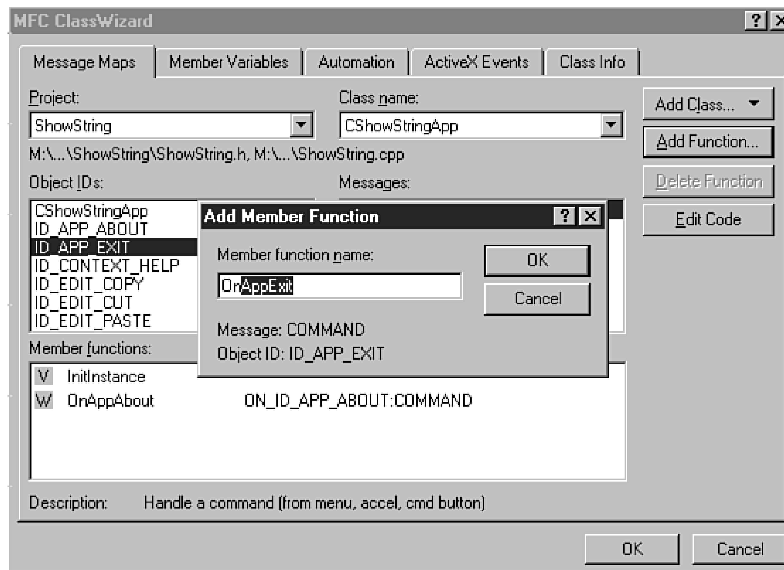


FIG. 3.5 It's possible, but not wise, to change the name for your command handler or command update handler from the name suggested by ClassWizard.

Not every command needs an update handler. The framework does some very nice work graying and ungraying for you automatically. Say you have a menu item—Network, Send—whose command is caught by the document. When there is no open document, this menu item is grayed by the framework, without any coding on your part. For many commands, it's enough that an object that can handle them exists, and no special updating is necessary. For others, you may want to check that something is selected or highlighted or that no errors are present before making certain commands available. That's when you use command updating. If you'd like to see an example of command updating at work, there's one in Chapter 8 in the "Command Updating" section.

UNIT - II

DOCUMENTS AND VIEWS

In This Chapter

Understanding the Document Class

Understanding the View Class

Creating the Rectangles Application

Other View Classes

Document Templates, Views, and Frame

Windows

Understanding the Document Class

When you generate your source code with AppWizard, you get an application featuring all the bells and whistles of a commercial 32-bit Windows application, including a toolbar, a status bar, ToolTips, menus, and even an About dialog box. However, in spite of all those features, the application really doesn't do anything useful. In order to create an application that does more than look pretty on your desktop, you need to modify the code that AppWizard generates. This task can be easy or complex, depending on how you want your application to look and act.

Probably the most important set of modifications are those related to the *document*—the information the user can save from your application and restore later—and to the *view*—the way that information is presented to the user. MFC's document/view architecture separates an application's data from the way the user actually views and manipulates that data. Simply, the document object is responsible for storing, loading, and saving the data, whereas the view object (which is just another type of window) enables the user to see the data onscreen and to edit that data in a way that is appropriate to the application. In this chapter, you learn the basics of how MFC's document/view architecture works.

SDI and MDI applications created with AppWizard are document/view applications. That means that AppWizard generates a class for you derived from CDocument, and delegates certain tasks to this new document class. It also creates a view class derived from CView and delegates other tasks to your new view class. Let's look through an AppWizard starter application and see what you get.

Choose File, New, and select the Projects tab. Fill in the project name as **App1** and fill in an appropriate directory for the project files. Make sure that MFC AppWizard (exe) is selected. Click OK.

Move through the AppWizard dialog boxes, changing the settings to match those in the following table, and then click Next to continue:

Step 1: Multiple documents

Step 2: Don't change the defaults presented by AppWizard

Step 3: Don't change the defaults presented by AppWizard

Step 4: Deselect all check boxes except Printing and Print Preview

Step 5: Don't change the defaults presented by AppWizard

Step 6: Don't change the defaults presented by AppWizard

After you click Finish on the last step, the New project information box summarizes your work. Click OK to create the project. Expand the App1 classes in ClassView, and you see that six classes have been created: CAboutDlg, CApp1App, CApp1Doc, CApp1View, CChildFrame, and CMainframe.

CApp1Doc represents a document; it holds the application's document data. You add storage for the document by adding data members to the CApp1Doc class. To see how this works, look atListing 4.1, which shows the header file AppWizard creates for the CApp1Doc class.

Listing 4.1 APP1DOC.H—The Header File for the CApp1Doc Class

```
// App1Doc.h : interface of the CApp1Doc class

//

///////////////////////////////////////////////////////////

#if
!defined(AFX_APP1DOC_H__43BB481D_64AE_11D0_9AF3_0080C81A397C__INCLUD
ED_)

#define
AFX_APP1DOC_H__43BB481D_64AE_11D0_9AF3_0080C81A397C__INCLUDED_

#if _MSC_VER > 1000

#pragma once

#endif // _MSC_VER > 1000

class CApp1Doc : public CDocument

{

protected: // create from serialization only

CApp1Doc();
```

```
DECLARE_DYNCREATE(CApp1Doc)
// Attributes
public:
// Operations
public:
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CApp1Doc)
public:
virtual BOOL OnNewDocument();
virtual void Serialize(CArchive& ar);
//}}AFX_VIRTUAL
// Implementation
public:
virtual ~CApp1Doc();
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif
protected:
// Generated message map functions
protected:
//{{AFX_MSG(CApp1Doc)
// NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
```

```
DECLARE_MESSAGE_MAP()

};

        ///////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}

// Microsoft Visual C++ will insert additional declarations

// immediately before the previous line.

#endif // !defined(AFX_APP1DOC_H__43BB481D_64AE_11D0_9AF3

        [ccc] _0080C81A397C__INCLUDED_)
```

Near the top of the listing, you can see the class declaration's Attributes section, which is followed by the public keyword. This is where you declare the data members that will hold your application's data. In the program that you create a little later in this chapter, the application must store an array of CPoint objects as the application's data. That array is declared as a member of the document class like this:

```
// Attributes

public:

CPoint points[100];
```

CPoint is an MFC class that encapsulates the information relevant to a point on the screen, most importantly the x and y coordinates of the point.

Notice also in the class's header file that the CApp1Doc class includes two virtual member functions called OnNewDocument() and Serialize(). MFC calls the OnNewDocument() function whenever the user selects the File, New command (or its toolbar equivalent, if a New button has been implemented in the application). You can use this function to perform whatever initialization must be performed on your document's data. In an SDI application, which has only a single document open at any time, the open document is closed and a new blank document is loaded into the same object; in an MDI application, which can have multiple documents open, a blank document is opened in addition to the documents that are already open. The Serialize() member function is where the document class loads and saves its data. This is discussed in Chapter 7, "Persistence and File I/O."

Understanding the View Class

As mentioned previously, the view class displays the data stored in the document object and enables the user to modify this data. The view object keeps a pointer to the document object, which it uses to access the document's member variables in order to display or modify them. Listing 4.2 is the header file for Capp1View, as generated by AppWizard.

Listing 4.2 APP1VIEW.H—The Header File for the CApp1View Class

```
// App1View.h : interface of the CApp1View class

//

///////////////////////////////////////////////////////////////////////////
/////

#if !defined(AFX_APP1VIEW_H__43BB481F_64AE_11D0_9AF3

[ccc]_0080C81A397C__INCLUDED_)

#define
AFX_APP1VIEW_H__43BB481F_64AE_11D0_9AF3_0080C81A397C__INCLUDED_

#if _MSC_VER > 1000

#pragma once

#endif // _MSC_VER > 1000

class CApp1View : public CView

{

protected: // create from serialization only

CApp1View();

DECLARE_DYNCREATE(CApp1View)

// Attributes

public:

CApp1Doc* GetDocument();

// Operations

public:

// Overrides

// ClassWizard generated virtual function overrides

//{{AFX_VIRTUAL(CApp1View)
```

```
public:
virtual void OnDraw(CDC* pDC); // overridden to draw this view
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
//}}AFX_VIRTUAL
// Implementation
public:
virtual ~CApp1View();
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif
protected:
// Generated message map functions
protected:
//{{AFX_MSG(CApp1View)
    // NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
#ifndef _DEBUG // debug version in App1View.cpp
inline CApp1Doc* CApp1View::GetDocument()
```

{ return (CApp1Doc*)m_pDocument; }

#endif

//
/////

//{{AFX_INSERT_LOCATION}}

// Microsoft Visual C++ will insert additional declarations

// immediately before the previous line.

#endif // !defined(AFX_APP1VIEW_H__43BB481F_64AE_11D0_9AF3

[ccc] _0080C81A397C__INCLUDED_)

Near the top of the listing, you can see the class's public attributes, where it declares the GetDocument() function as returning a pointer to a CApp1Doc object. Anywhere in the view class that you need to access the document's data, you can call GetDocument() to obtain a pointer to the document. For example, to add a CPoint object to the aforementioned array of CPoint objects stored as the document's data, you might use the following line:

GetDocument()->m_points[x] = point;

You also can do this a little differently, of course, by storing the pointer returned by

GetDocument() in a local pointer variable and then using that pointer variable to access the document's data, like this:

pDoc = GetDocument();

pDoc->m_points[x] = point;

The second version is more convenient when you need to use the document pointer in several places in the function, or if using the less clear GetDocument() ->variable version makes the code hard to understand.

Notice that the view class, like the document class, overrides a number of virtual functions from its base class. As you'll soon see, the OnDraw() function, which is the most important of these virtual functions, is where you paint your window's display. As for the other functions, MFC calls PreCreateWindow() before the window element (that is, the actual Windows window) is created and attached to the MFC window class, giving you a chance to modify the window's attributes (such as size and position). These two functions are discussed in more detail in Chapter 5, "Drawing on the Screen." OnPreparePrinting() is used to modify the Print dialog box before it displays for the user; the OnBeginPrinting() function gives you a chance to create GDI objects like pens and brushes that you need to handle the print job; and OnEndPrinting() is where you can destroy any objects you might have created in

OnBeginPrinting(). These three functions are discussed in Chapter 6, “Printing and Print Preview.”

Creating the Rectangles Application

Now that you’ve had an introduction to documents and views, a little hands-on experience should help you better understand how these classes work. In the steps that follow, you build the Rectangles application, which demonstrates the manipulation of documents and views. When you first run this application, it will draw an empty window. Wherever you click in the window, a small rectangle will be drawn. You can resize the window, or minimize and restore it, and the rectangles will be redrawn at all the coordinates where you clicked, because Rectangles keeps an array of coordinate points in the document and uses that array in the view.

First, use AppWizard to create the basic files for the Rectangles program, selecting the options listed in the following table. (AppWizard is first discussed in Chapter 1, “Building Your First Windows Application.” When you’re done, the New Project Information dialog box appears; it should look like Figure 4.1. Click the OK button to create the project files. Dialog Box Name Options to Select New Project Name the project **recs** and set the project path to the directory into which you want to store the project’s files. Leave the other options set to their defaults.

Step 1 Select Single Document.

Step 2 of 6 Leave default settings.

Step 3 of 6 Leave default settings.

Step 4 of 6 Turn off all application features except Printing and Print Preview.

Step 5 of 6 Leave default settings.

Step 6 of 6 Leave default settings.

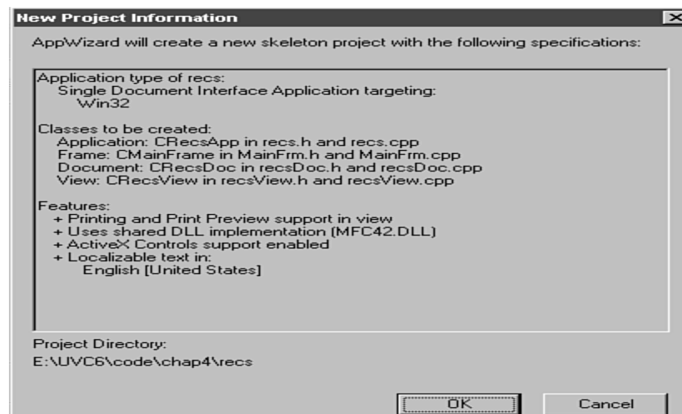


FIG. 4.1 When you create an SDI application with AppWizard, the project information summary confirms your settings.

Now that you have a starter application, it's time to add code to the document and view classes in order to create an application that actually does something. This application will draw many rectangles in the view and save the coordinates of the rectangles in the document. Follow these steps to add the code that modifies the document class to handle the application's data, which is an array of CPoint objects that determine where rectangles should be drawn in the view window:

1. Click the ClassView tab to display the ClassView in the project workspace window at the left of the screen.

2. Expand the recs classes by clicking the + sign before them.

3. Right-click the CRecsDoc class and choose Add Member Variable from the shortcut menu that appears.

4. Fill in the Add Member Variable dialog box. For Variable Type, enter CPoint. For Variable Name, enter m_points[100]. Make sure the Public radio button is selected. Click OK.

5. Again, right-click the CRecsDoc class and choose Add Member Variable.

6. For Variable Type, enter UINT. For Variable Name, enter m_pointIndex. Make sure the Public radio button is selected. Click OK.

7. Click the + next to CRecsDoc in ClassView to see the member variables and functions. The two member variables you added are now listed.

The m_points[] array holds the locations of rectangles displayed in the view window. The m_pointIndex data member holds the index of the next empty element of the array.

Now you need to get these variables initialized to appropriate values and then use them to draw the view. MFC applications that use the document/view paradigm initialize document data in a function called OnNewDocument(), which is called automatically when the application first runs and whenever the user chooses File, New. The list of member variables and functions of CRecsDoc should still be displayed in ClassView. Double-click OnNewDocument() in that list to edit the code. Using Listing 4.3 as a guide, remove the comments left by AppWizard and initialize m_pointIndex to zero.

Listing 4.3 RECSDOC.CPP—CRecsDoc::OnNewDocument()

```
BOOL CRecsDoc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
return FALSE;
m_pointIndex = 0;
return TRUE;
}
```

There is no need to initialize the array of points because the index into the array will be used to ensure no code tries to use an uninitialized element of the array. At this point your modifications to the document class are complete. As you'll see in Chapter 7, there are a few simple changes to make if you want this information actually saved in the document. In order to focus on the way documents and views work together, you will not be making those changes to the recs application.

Now turn your attention to the view class. It will use the document data to draw rectangles onscreen. A full discussion of the way that drawing works must wait for Chapter 5. For now it is enough to know that the OnDraw() function of your view class does the drawing. Expand the CRecsView class in ClassView and double-click OnDraw(). Using Listing 4.4 as a guide, remove the comments left by AppWizard and add code to draw a rectangle at each point in the array.

Listing 4.4 RECSVIEW.CPP—CRecsView::OnDraw()

```
void CRecsView::OnDraw(CDC* pDC)

{

CRecsDoc* pDoc = GetDocument();

ASSERT_VALID(pDoc);

UINT pointIndex = pDoc->m_pointIndex;

for (UINT i=0; i<pointIndex; ++i)

{

UINT x = pDoc->m_points[i].x;

UINT y = pDoc->m_points[i].y;

pDC->Rectangle(x, y, x+20, y+20);

}

}
```

Your modifications to the starter application generated by AppWizard are almost complete. You have added member variables to the document, initialized those variables in the document's OnNewDocument() function, and used those variables in the view's OnDraw() function. All that remains is to enable the user to add points to the array. As discussed in Chapter 3, "Messages and Commands," you catch the mouse message with ClassWizard and then add code to the message handler. Follow these steps:

1. Choose View, ClassWizard. The ClassWizard dialog box appears.

2. Make sure that CRecsView is selected in the Class Name and Object IDs boxes. Then, double-click WM_LBUTTONDOWN in the Messages box to add the OnLButtonDown() messageresponse function to the class. Whenever the application receives a WM_LBUTTONDOWN message, it will call OnLButtonDown().

3. Click the Edit Code button to jump to the OnLButtonDown() function in your code. Then, add the code shown in Listing 4.5 to the function.

Listing 4.5 RECSVIEW.CPP—CRecsView::OnLButtonDown()

```
void CRecsView::OnLButtonDown(UINT nFlags, CPoint point)

{

CRecsDoc *pDoc = GetDocument();

// don't go past the end of the 100 points allocated

if (pDoc->m_pointIndex == 100)

return;

//store the click location

pDoc->m_points[pDoc->m_pointIndex] = point;

pDoc->m_pointIndex++;

pDoc->SetModifiedFlag();

Invalidate();

CView::OnLButtonDown(nFlags, point);

}
```

The new OnLButtonDown() adds a point to the document's point array each time the user clicks the left mouse button over the view window. It increments m_pointIndex so that the next click goes into the point on the array after this one.

The call to SetModifiedFlag() marks this document as modified, or "dirty." MFC automatically prompts the user to save any dirty files on exit. (The details are found in Chapter 7.) Any code you write that changes any document variables should call SetModifiedFlag().

Finally, the call to Invalidate() causes MFC to call the OnDraw() function, where the window's display is redrawn with the new data. Invalidate() takes a single parameter (with the default value TRUE) that determines if the background is erased before calling OnDraw(). On rare occasions you may choose to call Invalidate(FALSE) so that OnDraw() draws over whatever was already onscreen.

Finally, a call to the base class OnLButtonDown() takes care of the rest of the work involved in handling a mouse click. You've now finished the complete application. Click the toolbar's Build button, or choose Build, Build from the menu bar, to compile and link the application. After you have the Rectangles application compiled and linked, run it by choosing Build, Execute. When you do, you see the application's main window. Place your mouse pointer over the window's client area and click. A rectangle appears. Go ahead and keep clicking. You can place up to 100 rectangles in the window (see Figure 4.2).

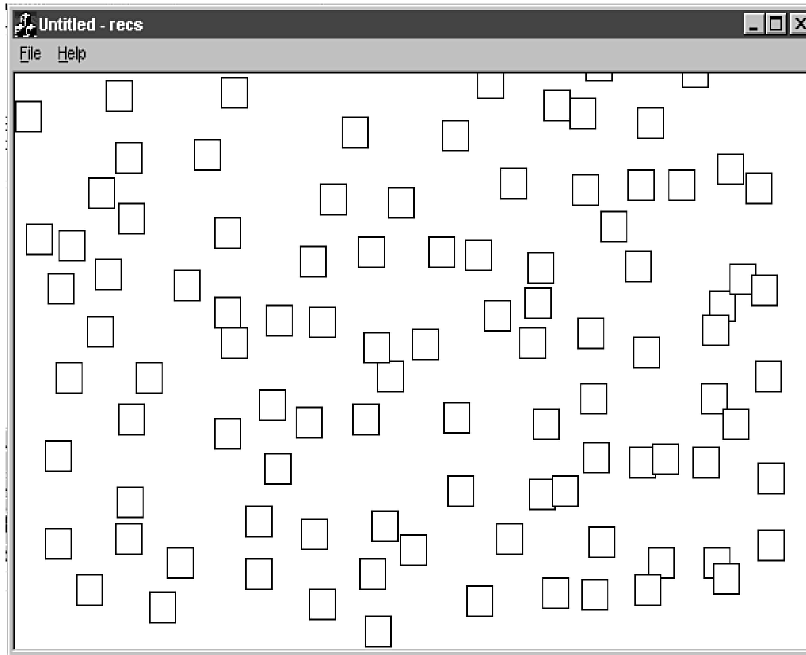


FIG. 4.2 The Rectangles application draws rectangles wherever you click.

Other View Classes

The view classes generated by AppWizard in this chapter's sample applications have been derived from MFC's CView class. There are cases, however, when it is to your advantage to derive your view class from one of the other MFC view classes derived from CView. These additional classes provide your view window with special capabilities such as scrolling and text editing. Table 4.1 lists the various view classes along with their descriptions

Table 4.1 View Classes

Class	**Description**
CView	The base view class from which the specialized view classes are derived
CCtrlView	A base class from which view classes that implement 32-bit Windows common controls (such as the ListView, TreeView, and RichEdit controls) are derived

Class	Description
CDaoRecordView	Same as CRecordView, except used with the OLE DB database classes
CEditView	A view class that provides basic text-editing features
CFormView	A view class that implements a form-like window using a dialog box resource
CHtmlView	A view class that can display HTML, with all the capabilities of Microsoft Internet Explorer
CListView	A view class that displays a ListView control in its window
COleDBRecordView	Same as CRecordView, except used with the DAO database classes
CRecordView	A view class that can display database records along with controls for navigating the database
CRichEditView	A view class that provides more sophisticated text-editing capabilities by using the RichEdit control
CScrollView	A view class that provides scrolling capabilities
CTreeView	A view class that displays a TreeView control in its window

To use one of these classes, substitute the desired class for the CView class in the application's project. When using AppWizard to generate your project, you can specify the view class you want in the wizard's Step 6 of 6 dialog box, as shown in Figure 4.3. When you have the desired class installed as the project's view class, you can use the specific class's member functions to control the view window. Chapter 5 demonstrates using the CScrollView class to implement a scrolling view.

A CEditView object, on the other hand, gives you all the features of a Windows edit control in your view window. Using this class, you can handle various editing and printing tasks, including find-and-replace. You can retrieve or set the current printer font by calling the GetPrinterFont() or SetPrinterFont() member function or get the currently selected text by calling GetSelectedText(). Moreover, the FindText() member function locates a given text string, and OnReplaceAll() replaces all occurrences of a given text string with another string.

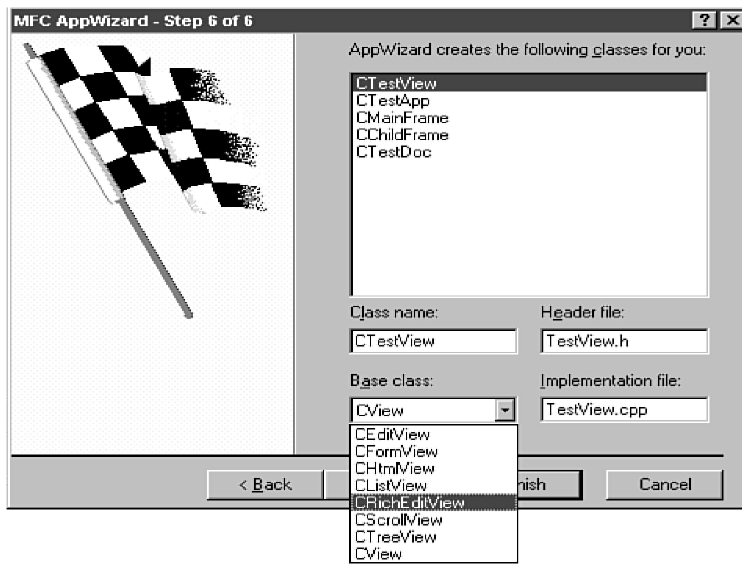


FIG. 4.3 You can use AppWizard to select your application's base view class.

The CRichEditView class adds many features to an edit view, including paragraph formatting (such as centered, right-aligned, and bulleted text), character attributes (including underlined, bold, and italic), and the capability to set margins, fonts, and paper size. As you might have guessed, the CRichEditView class features a rich set of methods you can use to control your application's view object.

Document Templates, Views, and Frame Windows

Because you've been working with AppWizard-generated applications in this chapter, you've taken for granted a lot of what goes on in the background of an MFC document/view program. That is, much of the code that enables the frame window (your application's main window), the document, and the view window to work together is automatically generated by AppWizard and manipulated by MFC.

For example, if you look at the InitInstance() method of the Rectangles application's CRecsApp class, you see (among other things) the lines shown in Listing 4.6.

Listing 4.6 RECS.CPP—Initializing an Application's Document

```
CSingleDocTemplate* pDocTemplate;

pDocTemplate = new CSingleDocTemplate(

IDR_MAINFRAME,

RUNTIME_CLASS(CRecsDoc),

RUNTIME_CLASS(CMainFrame),
```

RUNTIME_CLASS(CRecsView));

AddDocTemplate(pDocTemplate);

In Listing 4.6, you discover one secret that makes the document/view system work. In that code, the program creates a document-template object. These document templates have nothing to do with C++ templates, discussed in Chapter 26, "Exceptions and Templates." A document template is an older concept, named before C++ templates were implemented by Microsoft, that pulls together the following objects:

- A resource ID identifying a menu resource—IDR_MAINFRAME in this case
- A document class—CRecsDoc in this case
- A frame window class—always CMainFrame
- A view class—CRecsView in this case

Notice that you are not passing an object or a pointer to an object. You are passing the *name* of the class to a macro called RUNTIME_CLASS. It enables the framework to create instances of a class at runtime, which the application object must be able to do in a program that uses the document/view architecture. In order for this macro to work, the classes that will be created dynamically must be declared and implemented as such. To do this, the class must have the DECLARE_DYNCREATE macro in its declaration (in the header file) and the IMPLEMENT_DYNCREATE macro in its implementation. AppWizard takes care of this for you.

For example, if you look at the header file for the Rectangles application's CMainFrame class, you see the following line near the top of the class's declaration:

DECLARE_DYNCREATE(CMainFrame)

As you can see, the DECLARE_DYNCREATE macro requires the class's name as its single argument. Now, if you look near the top of CMainFrame's implementation file (MAINFRM.CPP), you see this line:

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

The IMPLEMENT_DYNCREATE macro requires as arguments the name of the class and the name of the base class.If you explore the application's source code further, you find that the document and view classes also contain the DECLARE_DYNCREATE and IMPLEMENT_DYNCREATE macros.

If you haven't heard of frame windows before, you should know that they contain all the windows involved in the applications—this means control bars as well as views. They also route messages and commands to views and documents, as discussed in Chapter 3.

The last line of Listing 4.6 calls AddDocTemplate() to pass the object on to the application object, CRecsApp, which keeps a list of documents. AddDocTemplate()

adds this document to this list and uses the document template to create the document object, the frame, and the view window.

Because this is a Single Document Interface, a single document template (CSingleDocTemplate) is created. Multiple Document Interface applications use one CMultiDocTemplate object for each kind of document they support. For example, a spreadsheet program might have two kinds of documents: tables and graphs. Each would have its own view and its own set of menus. Two instances of CMultiDocTemplate would be created in InitInstance(), each pulling together the menu, document, and view that belong together. If you've ever seen the menus in a program change as you switched from one view or document to another, you know how you can achieve the same effect: Simply associate them with different menu resource IDs as you build the document templates.

DRAWING ON THE SCREEN

In This Chapter

Understanding Device Contexts

Introducing the Paint1 Application

Building the Paint1 Application

Scrolling Windows

Building the Scroll Application

Understanding Device Contexts

Most applications need to display some type of data in their windows. You'd think that, because Windows is a device-independent operating system, creating window displays would be easier than luring a kitten with a saucer of milk. However, it's exactly Windows' device independence that places a little extra burden on a programmer's shoulders. Because you can never know in advance exactly what type of devices may be connected to a user's system, you can't make many assumptions about display capabilities. Functions that draw to the screen must do so indirectly through something called a *device context* (DC).

Although device independence forces you, the programmer, to deal with data displays indirectly, it helps you by ensuring that your programs run on all popular devices. In most cases, Windows handles devices for you through the device drivers that users have installed on the system. These device drivers intercept the data that the application needs to display and then translates the data appropriately for the device on which it will appear, whether that's a screen, a printer, or some other output device.

To understand how all this device independence works, imagine an art teacher trying to design a course of study appropriate for all types of artists. The teacher creates a course outline that stipulates the subject of a project, the suggested colors to be used, the dimensions of the finished project, and so on. What the teacher doesn't stipulate is the surface on which the project will be painted or the materials needed to paint on that surface. In other words, the teacher stipulates only general characteristics. The details of how these characteristics are applied to the finished project are left to each specific artist.

For example, an artist using oil paints will choose canvas as his drawing surface and oil paints, in the colors suggested by the instructor, as the paint. On the other hand, an artist using watercolors will select watercolor paper and will, of course, use watercolors instead of oils for paint. Finally, the charcoal artist will select the appropriate drawing surface for charcoal and will use a single color.

The instructor in this scenario is much like a Windows programmer. The programmer has no idea who may eventually use the program and what kind of system that user may have. The programmer can recommend the colors in which data should be displayed and the coordinates at which the data should appear, for example, but it's the device driver—the Windows artist— who ultimately decides how the data appears.

A system with a VGA monitor may display data with fewer colors than a system with a Super VGA monitor. Likewise, a system with a monochrome monitor displays the data in only a single color. High-resolution monitors can display more data than lower-resolution monitors. The device drivers, much like the artists in the imaginary art school, must take the display requirements and fine-tune them to the device on which the data will actually appear. And it's a data structure known as a *device context* that links the application to the device's driver.

A device context (DC) is little more than a data structure that keeps track of the attributes of a window's drawing surface. These attributes include the currently selected pen, brush, and font that will be used to draw onscreen. Unlike an artist, who can have many brushes and pens with which to work, a DC can use only a single pen, brush, or font at a time. If you want to use a pen that draws wider lines, for example, you need to create the new pen and then replace the DC's old pen with the new one. Similarly, if you want to fill shapes with a red brush, you must create the brush and *select it into the DC*, which is how Windows programmers describe replacing a tool in a DC.

A window's client area is a versatile surface that can display anything a Windows program can draw. The client area can display any type of data because everything displayed in a window— whether it be text, spreadsheet data, a bitmap, or any other type of data—is displayed graphically. MFC helps you display data by encapsulating Windows' GDI functions and objects into its DC classes.

Introducing the Paint1 Application

In this chapter, you will build the Paint1 application, which demonstrates fonts, pens, and brushes. Paint1 will use the document/view paradigm discussed in Chapter 4, "Documents and Views," and the view will handle displaying the data. When run, the application will display text in several different fonts. When users click the application, it displays lines drawn with several different pens. After another click, it displays boxes filled with a variety of brushes. The first step in creating Paint1 is to build an empty shell with AppWizard, as first discussed in Chapter 1, "Building Your First Windows Application." Choose File, New, and select the Projects tab. As shown in Figure 5.1, fill in the project name as **Paint1** and fill in an appropriate directory for the project files. Make sure that MFC AppWizard (exe) is selected. Click OK.

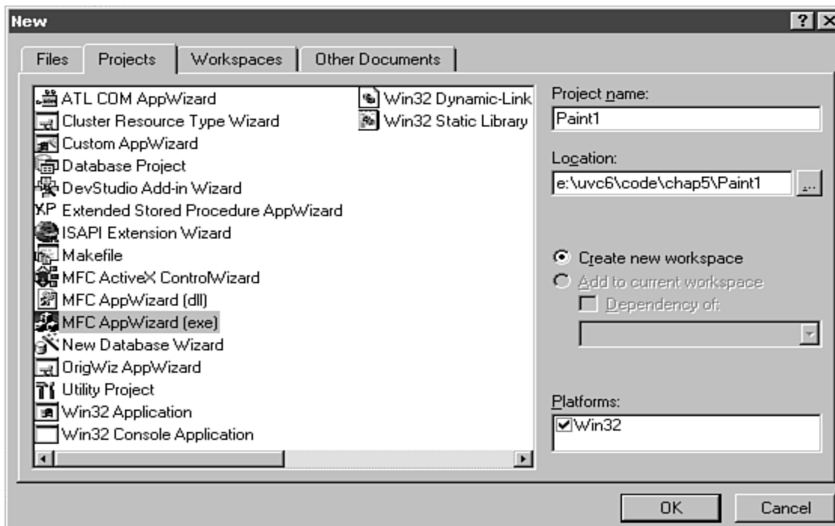


FIG. 5.1 Start an AppWizard project workspace called Paint1.

Move through the AppWizard dialog boxes, change the settings to match those in the list that follows, and then click Next to move to the next step.

Step 1: Select Single Document.

Step 2: Use default settings.

Step 3: Use default settings.

Step 4: Deselect all check boxes.

Step 5: Use default settings.

Step 6: Use default settings.

After you click Finish on the last step, the New Project Information box should resemble Figure 5.2. Click OK to create the project.

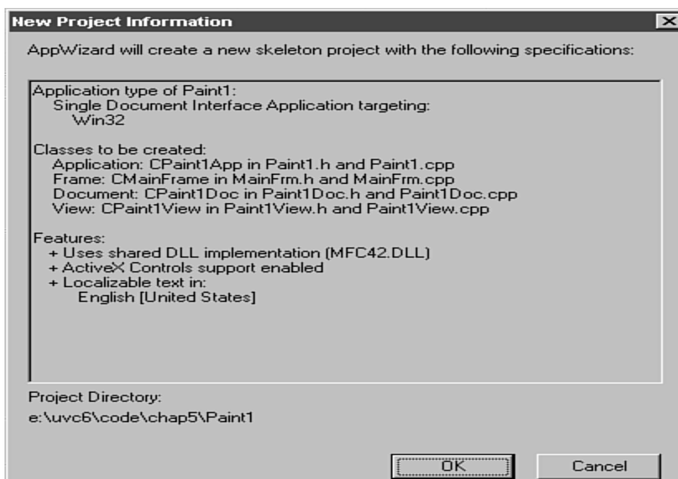


FIG. 5.2 The starter application for Paint1 is very simple.

Now that you have a starter application, it's time to add code to make it demonstrate some ways an MFC program can display data onscreen. By the time you get to the end of this chapter, the words *display context* won't make you scratch your head in perplexity.

Building the Paint1 Application

To build the Paint1 application, you first need to understand how painting and drawing work in an MFC program. Then you can set up the skeleton code to handle user clicks and the three different kinds of display. Finally, you'll fill in the code for each kind of display in turn.

Painting in an MFC Program

In Chapter 3, "Messages and Commands," you learned about message maps and how you can tell MFC which functions to call when it receives messages from Windows. One important message that every Windows program with a window must handle is WM_PAINT. Windows sends the WM_PAINT message to an application's window when the window needs to be redrawn. Several events cause Windows to send a WM_PAINT message:

- When users simply run the program: In a properly written Windows application, the application's window receives a WM_PAINT message almost immediately after being run, to ensure that the appropriate data is displayed from the very start.

- When the window has been resized or has recently been uncovered (fully or partially) by another window: Part of the window that wasn't visible before is now onscreen and must be updated.

- When a program indirectly sends itself a WM_PAINT message by invalidating its client area: This capability ensures that an application can change its window's contents almost any time it wants. For example, a word processor might invalidate its window after users paste some text from the Clipboard.

When you studied message maps, you learned to convert a message name to a message-map macro and function name. You now know, for example, that the message-map macro for a WM_PAINT message is ON_WM_PAINT(). You also know that the matching message-map function should be called OnPaint(). This is another case where MFC has already done most of the work of matching a Windows message with its message-response function. (If all this messagemap stuff sounds unfamiliar, you might want to review Chapter 3.)

You might guess that your next step is to catch the WM_PAINT message or to override the OnPaint() function that your view class inherited from CView, but you won't do that. Listing 5.1 shows the code for CView::OnPaint(). As you can see, WM_PAINT is already caught and handled for you.

Listing 5.1 CView::OnPaint()

```
void CView::OnPaint()

{

        // standard paint routine

        CPaintDC dc(this);

        OnPrepareDC(&dc);

        OnDraw(&dc);

}
```

CPaintDC is a special class for managing *paint DCs*—device contexts used only when responding to WM_PAINT messages. An object of the CPaintDC class does more than just create a DC; it also calls the BeginPaint() Windows API function in the class's constructor and calls EndPaint() in its destructor. When a program responds to WM_PAINT messages, calls to BeginPaint() and EndPaint() are required. The CPaintDC class handles this requirement without your having to get involved in all the messy details. As you can see, the CPaintDC constructor takes a single argument, which is a pointer to the window for which you're creating the DC. The this pointer points to the current view, so it's passed to the constructor to make a DC for the current view. OnPrepareDC() is a CView function that prepares a DC for use. You'll learn more about it in Chapter 6, "Printing and Print Preview." OnDraw() does the actual work of visually representing the document. In most cases you will write the OnDraw() code for your application and never touch OnPaint().

Switching the Display

The design for Paint1 states that when you click the application's window, the window's display changes. This seemingly magical feat is actually easy to accomplish. You add a member variable to the view to store what kind of display is being done and then change it when users click the window. In other words, the program routes WM_LBUTTONDOWN messages to the OnLButtonDown() message-response function, which sets the m_display flag as appropriate. First, add the member variable. You must add it by hand rather than through the shortcut menu because the type includes an enum declaration. Open Paint1View.h from the FileView and add these lines after the //Attributes comment:

```
protected:

enum {Fonts, Pens, Brushes} m_Display;
```

Choose ClassView in the Project Workspace pane, expand the classes, expand CPaint1View, and then double-click the constructor CPaint1View(). Add this line of code in place of the TODO comment:

```
m_Display = Fonts;
```

This initializes the display selector to the font demonstration. You use the display selector in the OnDraw() function called by CView::OnPaint(). AppWizard has created CPaint1View::OnDraw(), but it doesn't do anything at the moment. Double-click the function name in ClassView and add the code in Listing 5.2 to the function, removing the TODO comment left by AppWizard.

Listing 5.2 CPaint1View::OnDraw()

```
void CPaint1View::OnDraw(CDC* pDC)
{
    CPaint1Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    switch (m_Display)
    {
        case Fonts:
        ShowFonts(pDC);
        break;
        case Pens:
        ShowPens(pDC);
        break;
        case Brushes:
        ShowBrushes(pDC);
        break;
    }
}
```

You will write the three functions ShowFonts(), ShowPens(), and ShowBrushes() in upcoming sections of this chapter. Each function uses the same DC pointer that was passed to OnDraw() by OnPaint(). Add them to the class now by following these steps:

1. Right-click the CPaint1View class in ClassView and select Add Member Function.

2. Enter void for the Function Type.

3. Enter ShowFonts(CDC* pDC) for the Function Declaration.

4. Change the access to protected. Click OK.

5. Repeat steps 1 through 4 for ShowPens(CDC* pDC) and ShowBrushes(CDC* pDC).

The last step in arranging for the display to switch is to catch left mouse clicks and write code in the message handler to change m_display. Right-click CPaint1View in the ClassView and select Add Windows Message Handler from the shortcut menu that appears. Double-click WM_LBUTTONDOWN in the New Windows Messages/ Events list box. ClassWizard adds a function called OnLButtonDown() to the view and adds entries to the message map so that this function will be called whenever users click the left mouse button over this view. Click Edit Existing to edit the OnLButtonDown() you just created, and add the code shown in Listing 5.3.

Listing 5.3 CPaint1View::OnLButtonDown()

```
void CPaint1View::OnLButtonDown(UINT nFlags, CPoint point)
{
    if (m_Display == Fonts)
    m_Display = Pens;
    else if (m_Display == Pens)
    m_Display = Brushes;
    else
    m_Display = Fonts
    Invalidate();
    CView::OnLButtonDown(nFlags, point);
}
```

As you can see, depending on its current value, m_display is set to the next display type in the series. Of course, just changing the value of m_display doesn't accomplish much; the program still needs to redraw the contents of its window. The call to Invalidate() tells Windows that all of the window needs to be repainted. This causes Windows to generate a WM_PAINT message for the window, which means that eventually OnDraw() will be called and the view will be redrawn as a font, pen, or brush demonstration.

Using Fonts

Changing the font used in a view is a technique you'll want to use in various situations. It's not as simple as you might think because you can never be sure that any given font is actually installed on the user's machine. You set up a structure that holds information about the font you want, attempt to create it, and then work with the font you actually have, which might not be the font you asked for.

A Windows font is described in the LOGFONT structure outlined in Table 5.1. The LOGFONT structure uses 14 fields to hold a complete description of the font. Many fields can be set to 0 or the default values, depending on the program's needs.

Table 5.1 LOGFONT Fields and Their Descriptions

Field	Description
lfHeight	Font height in logical units
lfWidth	Font width in logical units
lfEscapement	Angle at which to draw the text
lfOrientation	Character tilt in tenths of a degree
lfWeight	Font weight
lfItalic	A nonzero value indicates italics
lfUnderline	A nonzero value indicates an underlined font
lfStrikeOut	A nonzero value indicates a strikethrough font
lfCharSet	Font character set
lfOutPrecision	How to match requested font to actual font
lfClipPrecision	How to clip characters that run over clip area
lfQuality	Print quality of the font
lfPitchAndFamily	Pitch and font family
lfFaceName	Typeface name

Some terms in Table 5.1 need a little explanation. The first is *logical units*. How high is a font with a height of 8 logical units, for example? The meaning of a logical unit depends on the *mapping mode* you're using, as shown in Table 5.2. The default mapping mode is MM_TEXT, which means that one logical unit is equal to 1 pixel. Mapping modes are discussed in more detail in Chapter 6.

Table 5.2 Mapping Modes

Mode	**Unit**
MM_HIENGLISH	0.001 inch
MM_HIMETRIC	0.01 millimeter
MM_ISOTROPIC	Arbitrary
MM_LOENGLISH	0.01 inch
MM_LOMETRIC	0.1 millimeter
MM_TEXT	Device pixel
MM_TWIPS	1/1440 inch

Escapement refers to writing text along an angled line. *Orientation* refers to writing angled text along a flat line. The font weight refers to the thickness of the letters. A number of constants have been defined for use in this field: FW_DONTCARE, FW_THIN, FW_EXTRALIGHT, FW_ULTRALIGHT, FW_LIGHT, FW_NORMAL, FW_REGULAR, FW_MEDIUM, FW_SEMIBOLD, FW_DEMIBOLD, FW_BOLD, FW_EXTRABOLD, FW_ULTRABOLD, FW_BLACK, and FW_HEAVY. Not all fonts are available in all weights. Four character sets are available (ANSI_CHARSET, OEM_CHARSET, SYMBOL_CHARSET, and UNICODE_CHARSET), but for writing English text you'll almost always use ANSI_CHARSET. (Unicode is discussed in Chapter 28, "Future Explorations.") The last field in the LOGFONT structure is the face name, such as Courier or Helvetica. Listing 5.4 shows the code you need to add to the empty ShowFonts() function you created earlier.

Listing 5.4 CPaint1View::ShowFonts()

```
void CPaint1View::ShowFonts(CDC * pDC)
{
    // Initialize a LOGFONT structure for the fonts.
    LOGFONT logFont;
    logFont.lfHeight = 8;
    logFont.lfWidth = 0;
    logFont.lfEscapement = 0;
    logFont.lfOrientation = 0;
    logFont.lfWeight = FW_NORMAL;
    logFont.lfItalic = 0;
    logFont.lfUnderline = 0;
    logFont.lfStrikeOut = 0;
    logFont.lfCharSet = ANSI_CHARSET;
```

```
        logFont.lfOutPrecision = OUT_DEFAULT_PRECIS;
        logFont.lfClipPrecision = CLIP_DEFAULT_PRECIS;
        logFont.lfQuality = PROOF_QUALITY;
        logFont.lfPitchAndFamily = VARIABLE_PITCH | FF_ROMAN;
        strcpy(logFont.lfFaceName, "Times New Roman");
        // Initialize the position of text in the window.
        UINT position = 0;
        // Create and display eight example fonts.
        for (UINT x=0; x<8; ++x)
        {
                // Set the new font's height.
                logFont.lfHeight = 16 + (x * 8);
                // Create a new font and select it into the DC.
                CFont font;
                font.CreateFontIndirect(&logFont);
                CFont* oldFont = pDC->SelectObject(&font);
                // Print text with the new font.
                position += logFont.lfHeight;
                pDC->TextOut(20, position, "A sample font.");
                // Restore the old font to the DC.
                pDC->SelectObject(oldFont);
        }
}
```

ShowFonts()starts by setting up a Times Roman font 8 pixels high, with a width that best matches the height and all other attributes set to normal defaults. To show the many fonts displayed in its window, the Paint1 application creates its fonts in a for loop, modifying the value of the LOGFONT structure's lfHeight member each time through the loop, using the loop variable x to calculate the new font height:

```
logFont.lfHeight = 16 + (x * 8);
```

Because x starts at 0, the first font created in the loop will be 16 pixels high. Each time through the loop, the new font will be 8 pixels higher than the previous one. After setting the font's height, the program creates a CFont object and calls its CreateFontIndirect() function, which attempts to create a CFont object corresponding to the LOGFONT you created. It will change the LOGFONT to describe the CFont that was actually created, given the fonts installed on the user's machine.

After ShowFonts() calls CreateFontIndirect(), the CFont object is associated with a Windows font. Now you can select it into the DC. Selecting objects into device contexts is a crucial concept in Windows output programming. You can't use any graphical object, such as a font, directly; instead, you select it into the DC and then use the DC. You always save a pointer to the old object that was in the DC (the pointer is returned from the SelectObject() call) and use it to restore the device context by selecting the old object again when you're finished. The same function, SelectObject(), is used to select various objects into a device context: the font you're using in this section, a pen, a brush, or a number of other drawing objects.

After selecting the new font into the DC, you can use the font to draw text onscreen. The local variable position holds the vertical position in the window at which the next line of text should be printed. This position depends on the height of the current font. After all, if there's not enough space between the lines, the larger fonts will overlap the smaller ones. When Windows created the new font, it stored the font's height (most likely the height that you requested, but maybe not) in the LOGFONT structure's lfHeight member. By adding the value stored in lfHeight, the program can determine the next position at which to display the line of text. To make the text appear onscreen, ShowFonts() calls TextOut().

TextOut()'s first two arguments are the X and Y coordinates at which to print the text. The third argument is the text to print. Having printed the text, you restore the old font to the DC in case this is the last time through the loop.

Build the application and run it. It should resemble Figure 5.3. If you click the window, it will go blank because the ShowPens() routine doesn't draw anything. Click again and it's still blank, this time because the ShowBrushes() routine doesn't draw anything. Click a third time and you are back to the fonts screen.

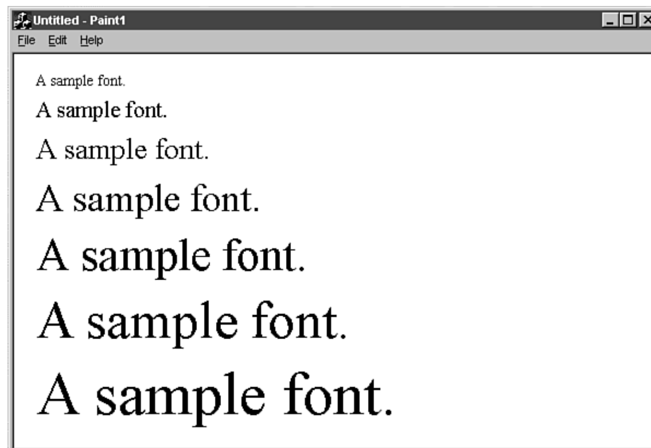


FIG. 5.3 The font display shows different types of text output

Sizing and Positioning the Window

As you can see in Figure 5.3, Paint1 doesn't display eight different fonts at 800´600 screen settings—only seven can fit in the window. To correct this, you need to

set the size of the window a little larger than the Windows default. In an MFC program, you do this in the mainframe class PreCreateWindow() function. This is called for you just before the mainframe window is created. The mainframe window surrounds the entire application and governs the size of the view.

PreCreateWindow() takes one parameter, a reference to a CREATESTRUCT structure. The CREATESTRUCT structure contains essential information about the window that's about to be created, as shown in Listing 5.5.

Listing 5.5 The CREATESTRUCT Structure

```
typedef struct tagCREATESTRUCT

{

    LPVOID lpCreateParams;

    HANDLE hInstance;

    HMENU hMenu;

    HWND hwndParent;

    int cy;

    int cx;

    int y;

    int x;

    LONG style;

    LPCSTR lpszName;

    LPCSTR lpszClass;

    DWORD dwExStyle;

} CREATESTRUCT;
```

If you've programmed Windows without application frameworks such as MFC, you'll recognize the information stored in the CREATESTRUCT structure. You used to supply much of this information when calling the Windows API function CreateWindow() to create your application's window. Of special interest to MFC programmers are the cx, cy, x, and y members of this structure. By changing cx and cy, you can set the window width and height, respectively. Similarly, modifying x and y changes the window's position. By overriding PreCreateWindow(), you have a chance to fiddle with the CREATESTRUCT structure before Windows uses it to create the window.

AppWizard created a CMainFrame::PreCreateWindow() function. Expand CMainFrame in ClassView, double-click PreCreateWindow() to edit it, and add lines to obtain the code shown in Listing 5.6. This sets the application's height and width. It also prevents users from resizing the application by using the bitwise and operator (&) to turn off the WS_SIZEBOX style bit.

Listing 5.6 CMainFrame::PreCreateWindow()

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)

{

    cs.cx = 440;

    cs.cy = 480;

    cs.style &= ~WS_SIZEBOX;

    if( !CFrameWnd::PreCreateWindow(cs) )

    return FALSE;

    return TRUE;

}
```

It's important that after your own code in PreCreateWindow(), you call the base class's PreCreateWindow(). Failure to do this will leave you without a valid window because MFC never gets a chance to pass the CREATESTRUCT structure on to Windows, so Windows never creates your window. When overriding class member functions, you usually need to call the base class's version. Build and run Paint1 to confirm that all eight fonts fit in the application's window. Now you're ready to demonstrate pens.

Using Pens

You'll be pleased to know that pens are much easier to deal with than fonts, mostly because you don't have to fool around with complicated data structures like LOGFONT. In fact, to create a pen, you need to supply only the pen's line style, thickness, and color. The Paint1 application's ShowPens() function displays in its window the lines drawn by using different pens created within a for loop. Listing 5.7 shows the code.

Listing 5.7 CPaint1View::ShowPens()

```
void CPaint1View::ShowPens(CDC * pDC)

{

    // Initialize the line position.
```

```
        UINT position = 10;

        // Draw sixteen lines in the window.

        for (UINT x=0; x<16; ++x)

        {

                // Create a new pen and select it into the DC.

                CPen pen(PS_SOLID, x*2+1, RGB(0, 0, 255));

                CPen* oldPen = pDC->SelectObject(&pen);

                // Draw a line with the new pen.

                position += x * 2 + 10;

                pDC->MoveTo(20, position);

                pDC->LineTo(400, position);

                // Restore the old pen to the DC.

                pDC->SelectObject(oldPen);

        }

}
```

Within the loop, ShowPens() first creates a custom pen. The constructor takes three parameters. The first is the line's style, one of the styles listed in Table 5.3. (You can draw only solid lines with different thicknesses. If you specify a pattern and a thickness greater than 1 pixel, the pattern is ignored and a solid line is drawn.) The second argument is the line thickness, which increases each time through the loop. The third argument is the line's color. The RGB macro takes three values for the red, green, and blue color components and converts them to a valid Windows color reference. The values for the red, green, and blue color components can be anything from 0 to 255—the higher the value, the brighter that color component. This code creates a bright blue pen. If all the color values were 0, the pen would be black; if the color values were all 255, the pen would be white.

Table 5.3 Pen Styles

Style	**Description**
PS_DASH	A pen that draws dashed lines
PS_DASHDOT	A pen that draws dash-dot patterned lines
PS_DASHDOTDOT	A pen that draws dash-dot-dot patterned lines

PS_DOT	A pen that draws dotted lines
PS_INSIDEFRAME	A pen that's used with shapes, in which the line's thickness must not extend outside the shape's frame
PS_NULL	A pen that draws invisible lines
PS_SOLID	A pen that draws solid lines

After creating the new pen, ShowPens() selects it into the DC, saving the pointer to the old pen. The MoveTo() function moves the pen to an X,Y coordinate without drawing as it moves; the LineTo() function moves the pen while drawing. The style, thickness, and color of the pen are used. Finally, you select the old pen into the DC.

Build and run Paint1 again. When the font display appears, click the window. You will see a pen display similar to the one in Figure 5.4.

Using Brushes

A pen draws a line of a specified thickness onscreen. A brush fills a shape onscreen. You can create solid and patterned brushes and even brushes from bitmaps that contain your own custom fill patterns. Paint1 will display both patterned and solid rectangles in the ShowBrushes() function, shown in Listing 5.8.

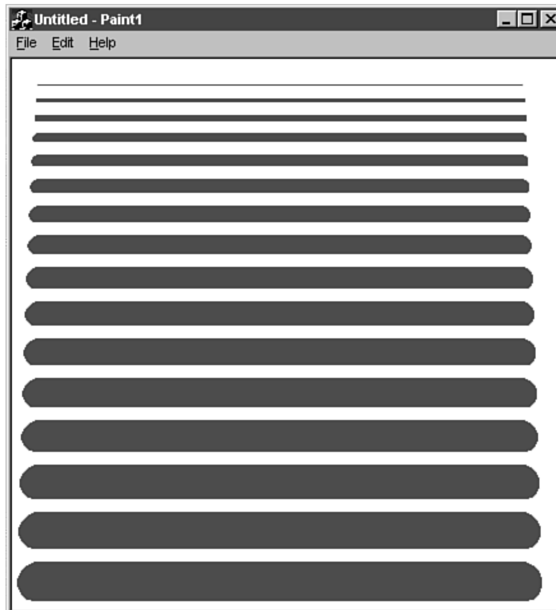


FIG. 5.4 The pen display shows the effect of setting line thickness.

Listing 5.8 CPaint1View::ShowBrushes()

```
void CPaint1View::ShowBrushes(CDC * pDC)

        // Initialize the rectangle position.
```

```
    UINT position = 0;
    // Select pen to use for rectangle borders
    CPen pen(PS_SOLID, 5, RGB(255, 0, 0));
    CPen* oldPen = pDC->SelectObject(&pen);
    // Draw seven rectangles.
    for (UINT x=0; x<7; ++x)
    {
        CBrush* brush;
        // Create a solid or hatched brush.
        if (x == 6)
        brush = new CBrush(RGB(0,255,0));
        else
        brush = new CBrush(x, RGB(0,160,0));

        // Select the new brush into the DC.
        CBrush* oldBrush = pDC->SelectObject(brush);
        // Draw the rectangle.
        position += 50;
        pDC->Rectangle(20, position, 400, position + 40);
        // Restore the DC and delete the brush.
        pDC->SelectObject(oldBrush);
        delete brush;
    }
    // Restore the old pen to the DC.
    pDC->SelectObject(oldPen);
}
```

The rectangles painted with the various brushes in this routine will all be drawn with a border. To arrange this, create a pen (this one is solid, 5 pixels thick, and bright red) and select it into the DC. It will be used to border the rectangles without any further work on your part. Like ShowFonts() and ShowPens(), this routine creates its graphical objects within a for loop. Unlike those two functions, ShowBrushes() creates a graphical object (in this routine, a brush) with a call to new. This enables you to call the one-argument constructor, which creates a solid brush, or the two-argument constructor, which creates a hatched brush. In Listing 5.8, the first argument to the two-argument constructor is just the loop variable, x. Usually, you don't want to show all the hatch patterns but want to select a specific one.

Use one of these constants for the hatch style:

- HS_HORIZONTAL—Horizontal
- HS_VERTICAL—Vertical
- HS_CROSS—Horizontal and vertical
- HS_FDIAGONAL—Forward diagonal
- HS_BDIAGONAL—Backward diagonal
- HS_DIAGCROSS—Diagonal in both directions

In a pattern that should be familiar by now, ShowBrushes() selects the brush into the DC, determines the position at which to work, uses the brush by calling Rectangle(), and then restores the old brush. When the loop is complete, the old pen is restored as well. Rectangle()is just one of the shape-drawing functions that you can call. Rectangle() takes as arguments the coordinates of the rectangle's upper-left and lower-right corners. Some others of interest are Chord(), DrawFocusRect(), Ellipse(), Pie(), Polygon(), PolyPolygon(), Polyline(), and RoundRect(), which draws a rectangle with rounded corners. Again, build and run Paint1. Click twice, and you will see the demonstration of brushes, as shown in Figure 5.5.

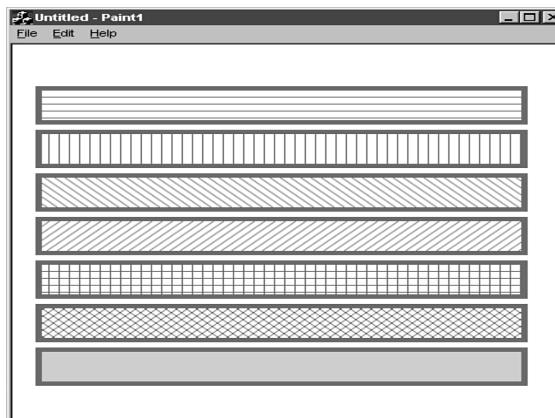


FIG. 5.5 The brushes display shows several patterns inside thick-bordered rectangles.

FIG. 5.6 Without erasing the background, the Paint1 application's windows appear messy.

Scrolling Windows

Those famous screen rectangles known as *windows* enable you to partition screen space between various applications and documents. Also, if a document is too large to completely fit within a window, you can view portions of it and scroll through it a bit at a time. The Windows operating system and MFC pretty much take care of the partitioning of screen space. However, if you want to enable users to view portions of a large document, you must create scrolling windows.

Adding scrollbars to an application from scratch is a complicated task. Luckily for Visual C++ programmers, MFC handles many of the details involved in scrolling windows over documents. If you use the document/view architecture and derive your view window from MFC's CScrollView class, you have scrolling capabilities almost for free. I say "almost" because you still must handle a few details, which you learn about in the following sections.

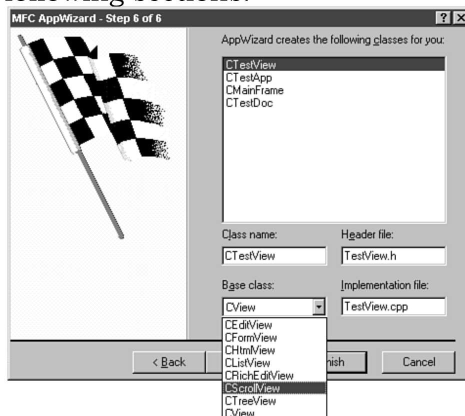


FIG. 5.7 You can create a scrolling window from within AppWizard.

Building the Scroll Application

In this section, you'll build a sample program called Scroll to experiment with a scrolling window. When Scroll first runs, it displays five lines of text. Each time you click the window, five lines of text are added to the display. When you have more lines of text than fit in the window, a vertical scrollbar appears, enabling you to scroll to the parts of the documents that you can't see.

As usual, building the application starts with AppWizard. Choose File, New, and select the Projects tab. Fill in the project name as **Scroll** and fill in an appropriate directory for the project files. Make sure that MFC AppWizard (exe) is selected. Click OK. Complete the AppWizard steps, selecting the following options:

Step 1: Select Single Document.

Step 2: Use default settings

Step 3: Use default settings.

Step 4: Deselect all check boxes

Step 5: Use default settings.

Step 6: Select CScrollView from the Base Class drop-

down box, as in Figure 5.7.

The New Project Information dialog box should resemble Figure 5.8. Click OK to create the project.

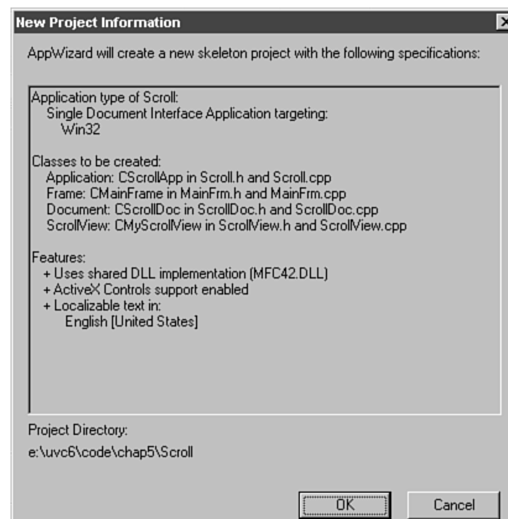


FIG. 5.8 Create a scroll application with AppWizard.

This application generates very simple lines of text. You need to keep track only of the number of lines in the scrolling view at the moment. To do this, add a variable to the document class by following these steps:

1. In ClassView, expand the classes and right-click CScrollDoc.

2. Choose Add Member Variable from the shortcut menu.

3. Fill in int as the variable type.

4. Fill in m_NumLines as the variable declaration.

5. Select Public for the Access.

Variables associated with a document are initialized in OnNewDocument(), as discussed in Chapter 4. In ClassView, expand CScrollDoc and double-click OnNewDocument() to expand it. Replace the TODO comments with this line of code:

```
m_NumLines = 5;
```

To arrange for this variable to be saved with the document and restored when the document is loaded, you must serialize it as discussed in Chapter 7, “Persistence and File I/O.” Edit CScrollDoc::Serialize() as shown in Listing 5.9.

Listing 5.9 CScrollDoc::Serialize()

```
void CScrollDoc::Serialize(CArchive& ar)

{

    if (ar.IsStoring())

    {

        ar << m_NumLines;

    }

    else

    {

        ar >> m_NumLines;

    }

}
```

Now all you need to do is use m_NumLines to draw the appropriate number of lines. Expand the view class, CMyScrollView, in ClassView and double-click OnDraw(). Edit it until it's the same as Listing 5.10. This is very similar to the ShowFonts() code from the Paint1 application earlier in this chapter.

Listing 5.10 CMyScrollView::OnDraw()

```
void CMyScrollView::OnDraw(CDC* pDC)

{

    CScrollDoc* pDoc = GetDocument();

    ASSERT_VALID(pDoc);



    // get the number of lines from the document

    int numLines = pDoc->m_NumLines;



    // Initialize a LOGFONT structure for the fonts.

    LOGFONT logFont;

    logFont.lfHeight = 24;

    logFont.lfWidth = 0;

    logFont.lfEscapement = 0;

    logFont.lfOrientation = 0;

    logFont.lfWeight = FW_NORMAL;

    logFont.lfItalic = 0;

    logFont.lfUnderline = 0;

    logFont.lfStrikeOut = 0;

    logFont.lfCharSet = ANSI_CHARSET;

    logFont.lfOutPrecision = OUT_DEFAULT_PRECIS;

    logFont.lfClipPrecision = CLIP_DEFAULT_PRECIS;

    logFont.lfQuality = PROOF_QUALITY;
```

```
logFont.lfPitchAndFamily = VARIABLE_PITCH | FF_ROMAN;
strcpy(logFont.lfFaceName, "Times New Roman");

// Create a new font and select it into the DC.
CFont* font = new CFont();
font->CreateFontIndirect(&logFont);
CFont* oldFont = pDC->SelectObject(font);

// Initialize the position of text in the window.
UINT position = 0;

// Create and display eight example lines.
for (int x=0; x<numLines; ++x)
{
        // Create the string to display.
        char s[25];
        wsprintf(s, "This is line #%d", x+1);

        // Print text with the new font.
        pDC->TextOut(20, position, s);
        position += logFont.lfHeight;
}

// Restore the old font to the DC, and
// delete the font the program created.
pDC->SelectObject(oldFont);
```

```
    delete font;

}
```

Build and run the Scroll application. You will see a display similar to that in Figure 5.9. No scrollbars appear because all the lines fit in the window.

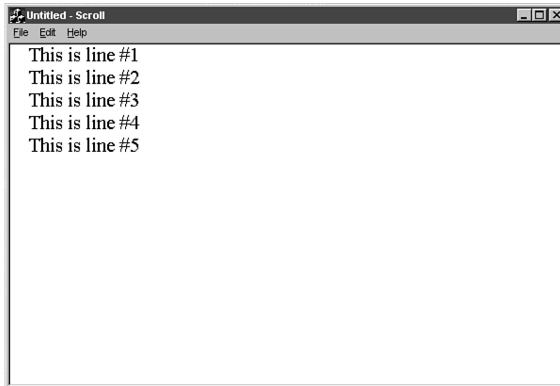


FIG. 5.9 At first, the scroll application displays five lines of text and no scrollbars.

Adding Code to Increase Lines

To increase the number of lines whenever users click the window, you need to add a message handler to handle left mouse clicks and then write the code for the handler. Right-click CMyScrollView in ClassView and choose Add Windows Message Handler. Double-click WM_LBUTTONDOWN to add a handler and click the Edit Existing button to change the code. Listing 5.11 shows the completed handler. It simply increases the number of lines and calls Invalidate() to force a redraw. Like so many message handlers, it finishes by passing the work on to the base class version of this function.

Listing 5.11 CMyScrollView::OnLButtonDown()

```
void CMyScrollView::OnLButtonDown(UINT nFlags, CPoint point)

{

    CScrollDoc* pDoc = GetDocument();

    ASSERT_VALID(pDoc);



    // Increase number of lines to display.

    pDoc->m_NumLines += 5;
```

```
        // Redraw the window.

        Invalidate();

        CScrollView::OnLButtonDown(nFlags, point);

}
```

Adding Code to Decrease Lines

So that you can watch scrollbars disappear as well as appear, why not implement a way for users to decrease the number of lines in the window? If left-clicking increases the number of lines, it makes sense that right-clicking would decrease it. Add a handler for WM_RBUTTONDOWN just as you did for WM_LBUTTONDOWN, and edit it until it's just like Listing 5.12. This function is a little more complicated because it ensures that the number of lines is never negative.

Listing 5.12 CMyScrollView::OnRButtonDown()

```
void CMyScrollView::OnRButtonDown(UINT nFlags, CPoint point)

{

        CScrollDoc* pDoc = GetDocument();

        ASSERT_VALID(pDoc);



        // Decrease number of lines to display.

        pDoc->m_NumLines -= 5;

        if (pDoc->m_NumLines < 0)

        {

                pDoc->m_NumLines = 0;

        }



        // Redraw the window.

        Invalidate();

        CScrollView::OnRButtonDown(nFlags, point);

}
```

If you build and run Scroll now and click the window, you can increase the number of lines, but scrollbars don't appear. You need to add some lines to OnDraw() to make that happen. Before you do, review the way that scrollbars work. You can click three places on a vertical scrollbar: the thumb (some people call it the elevator), above the thumb, or below it. Clicking the thumb does nothing, but you can click and hold to drag it up or down. Clicking above it moves you one page (screenful) up within the data. Clicking below it moves you one page down. What's more, the size of the thumb is a visual representation of the size of a page in proportion to the entire document. Clicking the up arrow at the top of the scrollbar moves you up one line in the document; clicking the down arrow at the bottom moves you down one line.

What all this means is that the code that draws the scrollbar and handles the clicks needs to know the size of the entire document, the page size, and the line size. You don't have to write code to draw scrollbars or to handle clicks on the scrollbar, but you do have to pass along some information about the size of the document and the current view. The lines of code you need to add to OnDraw() are in Listing 5.13; add them after the for loop and before the old font is selected back into the DC.

Listing 5.13 Lines to Add to OnDraw()

```
// Calculate the document size.

CSize docSize(100, numLines*logFont.lfHeight);



// Calculate the page size.

        CRect rect;

GetClientRect(&rect);

CSize pageSize(rect.right, rect.bottom);



// Calculate the line size.

CSize lineSize(0, logFont.lfHeight);



// Adjust the scrollers.

SetScrollSizes(MM_TEXT, docSize, pageSize, lineSize);
```

This new code must determine the document, page, and line sizes. The document size is the width and height of the screen area that could hold the entire document. This is calculated by using the number of lines in the entire document and the height of a line. (CSize is an MFC class created especially for storing the widths and heights of objects.) The page size is simply the size of the client rectangle of this view, and the line size is the height of the font. By setting the horizontal component of the line size to 0, you prevent horizontal scrolling.

These three sizes must be passed along to implement scrolling. Simply call SetScrollSizes(), which takes the mapping mode, document size, page size, and line size. MFC will set the scrollbars properly for any document and handle user interaction with the scrollbars.

Build and run Scroll again and generate some more lines. You should see a scrollbar like the one in Figure 5.10. Add even more lines and you will see the thumb shrink as the document size grows. Finally, resize the application horizontally so that the text won't all fit. Notice how no horizontal scrollbars appear, because you set the horizontal line size to 0.

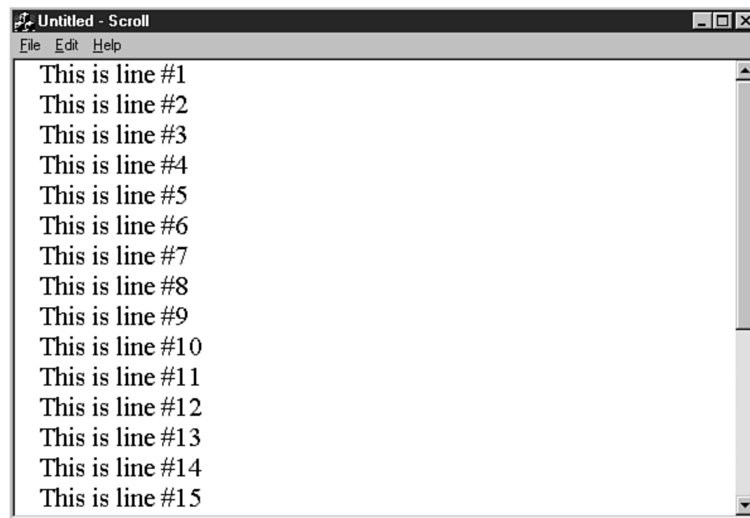


FIG. 5.10 After displaying more lines than fit in the window, the vertical scrollbar appears.

PRINTING AND PRINT PREVIEW

In this chapter

Understanding Basic Printing and Print Preview with MFC

Scaling

Printing Multiple Pages

Setting the Origin

MFC and Printing

Understanding Basic Printing and Print Preview with MFC

If you brought together 10 Windows programmers and asked them what part of creating Windows applications they thought was the hardest, probably at least half of them would choose printing documents. Although the device-independent nature of Windows makes it easier for users to get peripherals working properly, programmers must take up some of the slack by programming all devices in a general way. At one time, printing from a Windows application was a nightmare that only the most experienced programmers could handle. Now, however, thanks to application frameworks such as MFC, the job of printing documents from a Windows application is much simpler.

MFC handles so much of the printing task for you that, when it comes to simple one-page documents, you have little to do on your own. To see what I mean, follow these steps to create a basic MFC application that supports printing and print preview:

1. Choose File, New; select the Projects tab and start a new AppWizard project workspace called Print1 (see Figure 6.1).

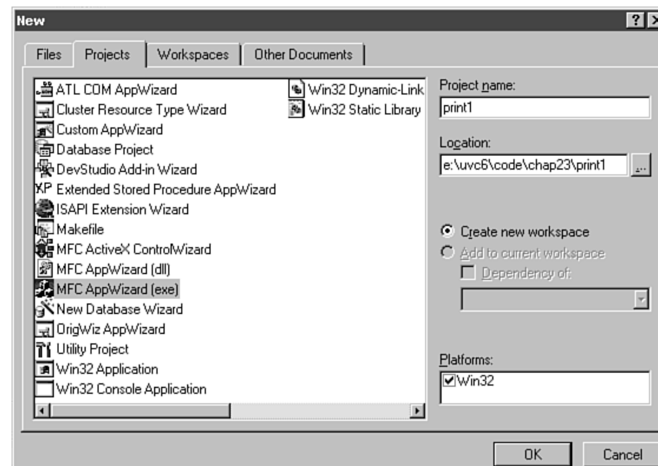


FIG. 6.1 Start an AppWizard project workspace called Print1.

2. Give the new project the following settings in the AppWizard dialog boxes. The New Project Information dialog box should then look like Figure 6.2.

Step 1: Choose Single Document.

Step 2: Don't change the defaults presented by AppWizard.

Step 3: Don't change the defaults presented by AppWizard.

Step 4: Turn off all features except Printing and Print Preview.

Step 5: Don't change the defaults presented by AppWizard.

Step 6: Don't change the defaults presented by

AppWizard.

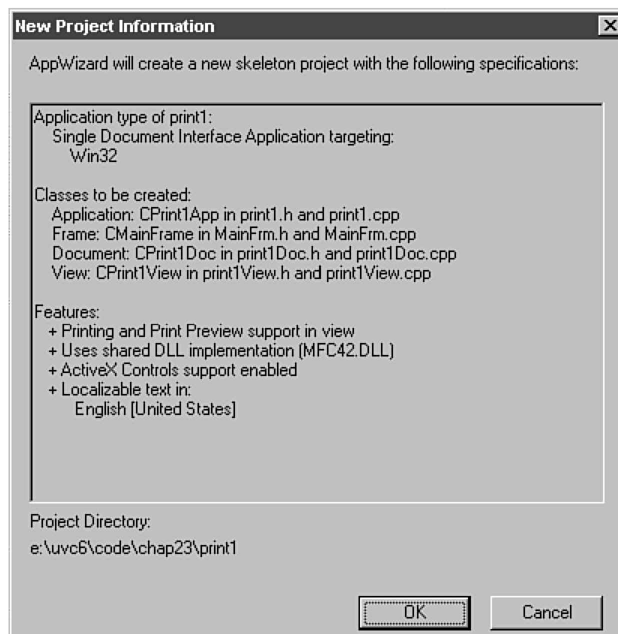


FIG. 6.2 The New Project Information dialog box.

Expand the classes in ClassView, expand CPrint1View, double-click the OnDraw() function, and add the following line of code to it, right after the comment TODO: adddraw code for native data here:

```
pDC->Rectangle(20, 20, 220, 220);
```

You've seen the Rectangle() function twice already: in the Recs app of Chapter 4, "Documents and Views," and the Paint1 app of Chapter 5, "Drawing on the Screen." Adding this function to the OnDraw() function of an MFC program's view class causes the program to draw a rectangle. This one is 200 pixels by 200 pixels, located 20 pixels down from the top of the view and 20 pixels from the left edge.

Believe it or not, you've just created a fully print-capable application that can display its data (a rectangle) not only in its main window but also in a print preview window and on the printer. To run the Print1 application, first compile and link the source code by choosing Build, Build or by pressing F7. Then, choose Build, Execute to run the program. You will see the window shown in Figure 6.3. This window contains the application's output data, which is simply a rectangle. Next, choose File, Print Preview. You see the print preview window, which displays the document as it will appear if you print it (see Figure 6.4). Go ahead and print the document (choose File, Print). These commands have been implemented for you because you chose support for printing and print preview when you created this application with AppWizard.

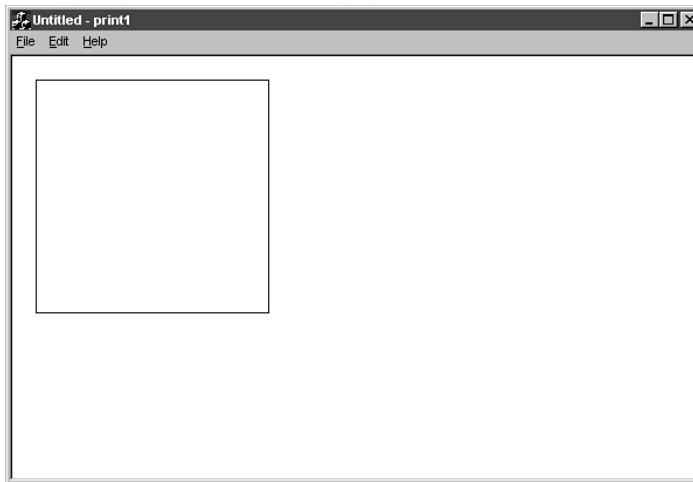


FIG. 6.3 Print1 displays a rectangle when you first run it.

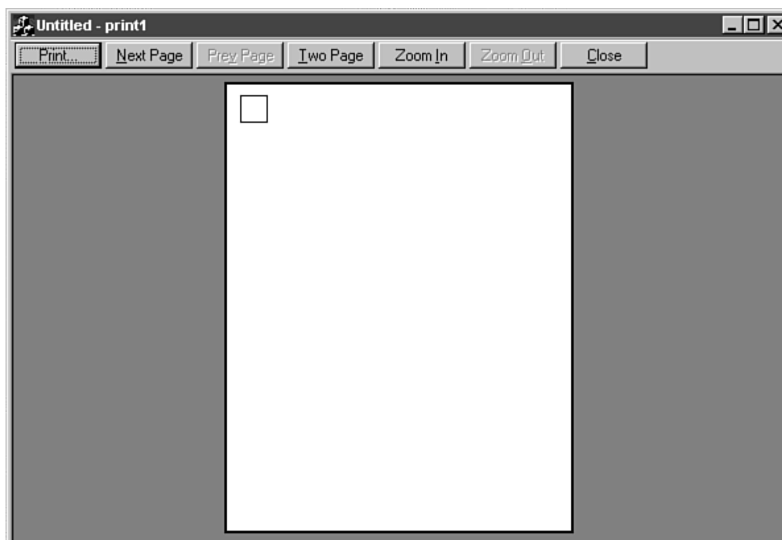


FIG. 6.4 The Print1 application automatically handles print previewing, thanks to the MFC AppWizard.

Scaling

One thing you may notice about the printed document and the one displayed onscreen is that, although the screen version of the rectangle takes up a fairly large portion of the application's window, the printed version is tiny. That's because the pixels onscreen and the dots on your printer are different sizes. Although the rectangle is 200 dots square in both cases, the smaller printer dots yield a rectangle that appears smaller. This is how the default Windows MM_TEXT graphics mapping mode works. If you want to scale the printed image to a specific size, you might want to choose a different mapping mode. Table 6.1 lists the mapping modes from which you can choose.

Table 6.1 Mapping Modes

Mode	Unit	X	Y
MM_HIENGLISH	0.001inch	Increases right	Increases up
MM_HIMETRIC	0.01 millimeter	Increases right	Increases up
MM_ISOTROPIC	User-defined	User-defined	User-defined
MM_LOENGLISH	0.01 inch	Increases right	Increases up
MM_LOMETRIC	0.1 millimeter	Increases right	Increases up
MM_TEXT	Device pixel	Increases right	Increases down
MM_TWIPS	1/1440 inch	Increases right	Increases up

Working with graphics in MM_TEXT mode causes problems when printers and screens can accommodate a different number of pixels per page. A better mapping mode for working with graphics is MM_LOENGLISH, which uses a hundredth of an inch, instead of a dot or pixel, as a unit of measure. To change the Print1 application so that it uses the MM_LOENGLISH mapping mode, replace the line you added to the OnDraw() function with the following two lines:

```
pDC->SetMapMode(MM_LOENGLISH);

pDC->Rectangle(20, -20, 220, -220);
```

The first line sets the mapping mode for the device context. The second line draws the rectangle by using the new coordinate system. Why the negative values? If you look at MM_LOENGLISH in Table 6.1, you see that although X coordinates increase to the right as you expect, Y coordinates increase upward rather than downward. Moreover, the default coordinates for the window are located in the lower-right quadrant of the Cartesian coordinate system, as shown in Figure 6.5. Figure 6.6 shows the print preview window when the application uses the MM_LOENGLISH

mapping mode. When you print the document, the rectangle is exactly 2 inches square because a unit is now 1/100 of an inch and the rectangle is 200 units square.

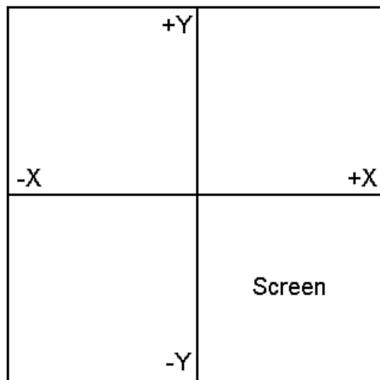


FIG. 6.5 The MM_LOENGLISH mapping mode's default coordinates derive from the

Cartesian coordinate system.

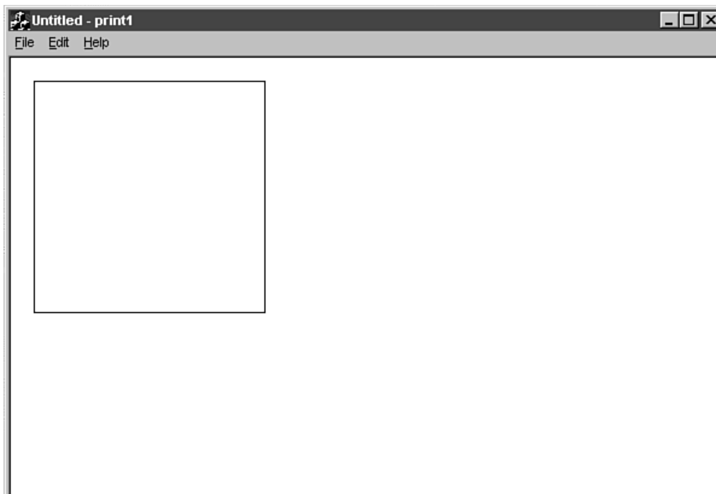


FIG. 6.6 The rectangle to be printed matches the rectangle onscreen when you use

MM_LOENGLISH as your mapping mode

Printing Multiple Pages

When your application's document is as simple as Print1's, adding printing and print previewing capabilities to the application is virtually automatic. This is because the document is only a single page and requires no pagination. No matter what you draw in the document window (except bitmaps), MFC handles all the printing tasks for you. Your view's OnDraw() function is used for drawing onscreen, printing to the printer, and drawing the print preview screen. Things become more complex, however,

when you have larger documents that require pagination or some other special handling, such as the printing of headers and footers.

To get an idea of the problems with which you're faced with a more complex document, modify Print1 so that it prints lots of rectangles—so many that they can't fit on a single page. This will give you an opportunity to deal with pagination. Just to make things more interesting, add a member variable to the document class to hold the number of rectangles to be drawn, and allow the users to increase or decrease the number of rectangles by left- or right-clicking.

Follow these steps:

1. Expand CPrint1Doc in ClassView, right-click it, and choose Add Member Variable from the shortcut menu. The variable type is int, the declaration is m_numRects, and the access should be public. This variable will hold the number of rectangles to display.

2. Double-click the CPrint1Doc constructor and add this line to it:

```
m_numRects = 5;
```

This line arranges to display five rectangles in a brand new document.

3. Use ClassWizard to catch mouse clicks (WM_LBUTTONDOWN messages) by adding an OnLButtonDown() function to the view class (see Figure 6.7).

4. Click the Edit Code button to edit the new OnLButtonDown() function. It should resemble Listing 6.1. Now the number of rectangles to be displayed increases each time users.

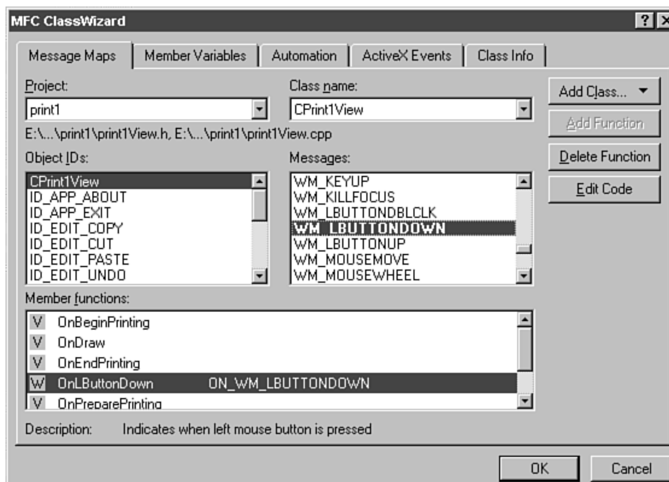


FIG. 6.7 Use ClassWizard to add the OnLButtonDown() function.

Listing 6.1 print1View.cpp —CPrint1View::OnLButtonDown()

```
void CPrint1View::OnLButtonDown(UINT nFlags, CPoint point)

{
```

```
        CPrint1Doc* pDoc = GetDocument();

        ASSERT_VALID(pDoc);

        pDoc->m_numRects++;

        Invalidate();

        CView::OnLButtonDown(nFlags, point);

}
```

5. Use ClassWizard to add the OnRButtonDown() function to the view class, as shown in Figure 6.8.

6. Click the Edit Code button to edit the new OnRButtonDown() function. It should resemble Listing 6.2. Now the number of rectangles to be displayed decreases each time users right-click.

Listing 6.2 print1View.cpp —CPrint1View::OnRButtonDown()

```
void CPrint1View::OnRButtonDown(UINT nFlags, CPoint point)

{

        CPrint1Doc* pDoc = GetDocument();

        ASSERT_VALID(pDoc);

        if (pDoc->m_numRects > 0)

        {

                pDoc->m_numRects--;

                Invalidate();

        }

        CView::OnRButtonDown(nFlags, point);

}
```

7. Rewrite the view's OnDraw() to draw many rectangles (refer to Listing 6.3). Print1 now draws the selected number of rectangles one below the other, which may cause the document to span multiple pages. It also displays the number of rectangles that have been added to the document.

Listing 6.3 print1View.cpp —CPrint1View::OnDraw()

```
void CPrint1View::OnDraw(CDC* pDC)
```

```
{
        CPrint1Doc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        // TODO: add draw code for native data here
        pDC->SetMapMode(MM_LOENGLISH);
        char s[10];
        wsprintf(s, "%d", pDoc->m_numRects);
        pDC->TextOut(300, -100, s);
        for (int x=0; x<pDoc->m_numRects; ++x)
        {
                pDC->Rectangle(20, -(20+x*200),
                200, -(200+x*200));
        }
}
```

When you run the application now, you see the window shown in Figure 6.9. The window not only displays the rectangles but also displays the rectangle count so that you can see how many rectangles you've requested. When you choose File, Print Preview, you see the print preview window. Click the Two Page button to see the window shown in Figure 6.10. The five rectangles display properly on the first page, with the second page blank.

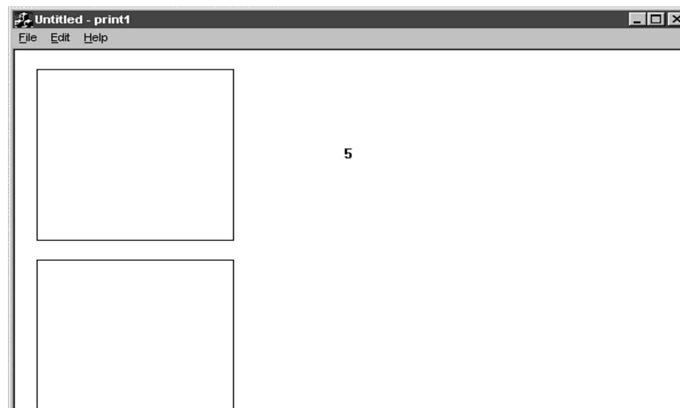


FIG. 6.9 Print1 now displays multiple rectangles.

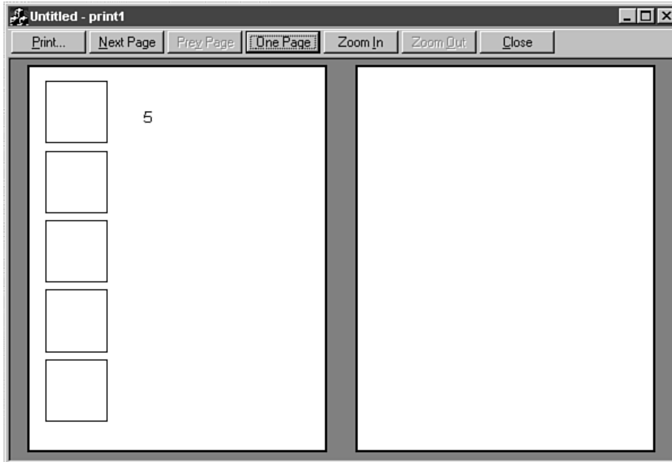


FIG. 6.10 Five rectangles are previewed properly; they will print on a single page.

Now, go back to the application's main window and click inside it three times to add three more rectangles. Right-click to remove one. (The rectangle count displayed in the window should be seven.) After you add the rectangles, choose File, Print Preview again to see the two-page print preview window. Figure 6.11 shows what you see. The program hasn't a clue how to print or preview the additional page. The sixth rectangle runs off the bottom of the first page, but nothing appears on the second page.

The first step is to tell MFC how many pages to print (or preview) by calling the SetMaxPage() function in the view class's OnBeginPrinting() function. AppWizard gives you a skeleton OnBeginPrinting() that does nothing. Modify it so that it resembles Listing 6.4.

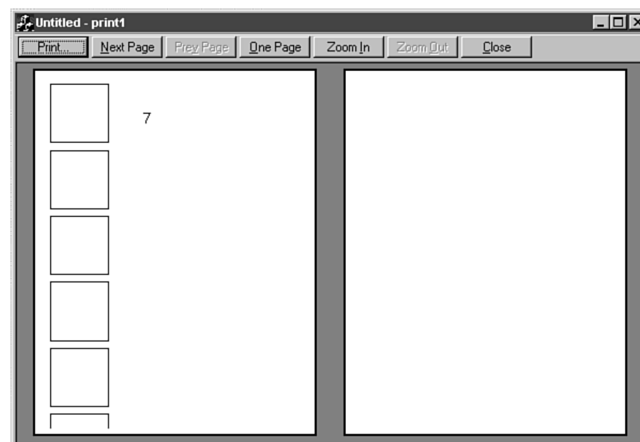


FIG. 6.11 Seven rectangles do not yet appear correctly on multiple pages

Listing 6.4 print1View.cpp —CPrint1View::OnBeginPrinting()

```
void CPrint1View::OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo)
{
    CPrint1Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    int pageHeight = pDC->GetDeviceCaps(VERTRES);
    int logPixelsY = pDC->GetDeviceCaps(LOGPIXELSY);
    int rectHeight = (int)(2.2 * logPixelsY);
    int numPages = pDoc->m_numRects * rectHeight / pageHeight + 1;
    pInfo->SetMaxPage(numPages);
}
```

OnBeginPrinting() takes two parameters: a pointer to the printer device context and a pointer to a CPrintInfo object. Because the default version of OnBeginPrinting() doesn't refer to these two pointers, the parameter names are commented out to avoid compilation warnings, like this:

```
void CPrint1View::OnBeginPrinting(CDC* /*pDC*/ , CPrintInfo* /*pInfo*/)
```

However, to set the page count, you need to access both the CDC and CPrintInfo objects, so your first task is to uncomment the function's parameters. Now you need to get some information about the device context (which, in this case, is a printer device context). Specifically, you need to know the page height (in single dots) and the number of dots per inch. You obtain the page height with a call to GetDeviceCaps(), which gives you information about the capabilities of the device context. You ask for the vertical resolution (the number of printable dots from the top of the page to the bottom) by passing the constant VERTRES as the argument. Passing HORZRES gives you the horizontal resolution. There are 29 constants you can pass to GetDeviceCaps(), such as NUMFONTS for the number of fonts that are supported and DRIVERVERSION for the driver version number. For a complete list, consult the online Visual C++ documentation.

Print1 uses the MM_LOENGLISH mapping mode for the device context, which means that the printer output uses units of 1/100 of an inch. To know how many rectangles will fit on a page, you have to know the height of a rectangle in dots so that you can divide dots per page by dots per rectangle to get rectangles per page. (You can see now why your application must know all about your document to calculate the page count.) You know that each rectangle is 2 inches high with 20/100 of an inch of space between each rectangle. The total distance from the start of one rectangle to the start of the next, then, is 2.2 inches. The call to GetDeviceCaps() with an argument of LOGPIXELSY gives the dots per inch of this printer; multiplying by 2.2 gives the dots

per rectangle. You now have all the information to calculate the number of pages needed to fit the requested number of rectangles. You pass that number to SetMaxPage(), and the new OnBeginPrinting() function is complete.

Again, build and run the program. Increase the number of rectangles to seven by clicking twice in the main window. Now choose File, Print Preview and look at the two-page print preview window (see Figure 6.12). Whoops! You obviously still have a problem somewhere. Although the application is previewing two pages, as it should with seven rectangles, it's printing exactly the same thing on both pages. Obviously, page two should take up where page one left off, rather than redisplay the same data from the beginning. There's still some work to do.

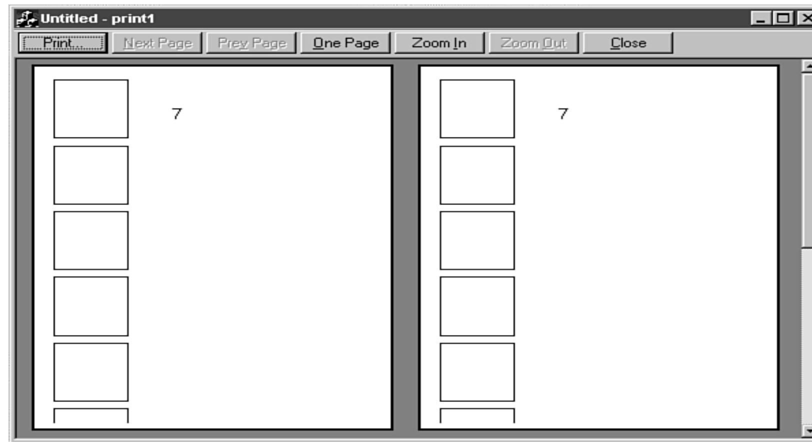


FIG. 6.12 The Print1 application still doesn't display multiple pages correctly.

Setting the Origin

To get the second and subsequent pages to print properly, you have to change where MFC believes the top of the page to be. Currently, MFC just draws the pages exactly as you told it to do in CPrint1View::OnDraw(), which displays all seven rectangles from the top of the page to the bottom. To tell MFC where the new top of the page should be, you first need to override the view class's OnPrepareDC() function. Bring up ClassWizard and choose the Message Maps tab. Ensure that CPrintView is selected in the Class Name box, as shown in Figure 6.13. Click CPrintView in the Object IDs box and OnPrepareDC in the Messages box, and then click Add Function. Click the Edit Code button to edit the newly added function. Add the code shown in Listing 6.5.

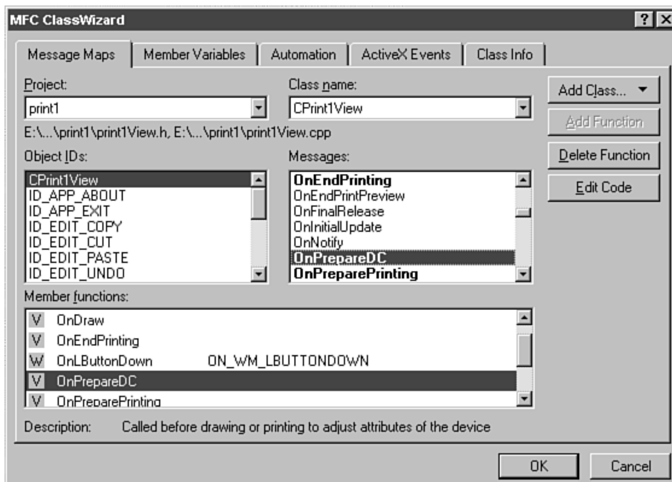


FIG. 6.13 Use ClassWizard to override the OnPrepareDC() function.

Listing 6.5 print1View.cpp —CPrint1View::OnPrepareDC()

```
void CPrint1View::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
        if (pDC->IsPrinting())
        {
                int pageHeight = pDC->GetDeviceCaps(VERTRES);
                int originY = pageHeight * (pInfo->m_nCurPage - 1);
                pDC->SetViewportOrg(0, -originY);
        }
        CView::OnPrepareDC(pDC, pInfo);
}
```

The MFC framework calls OnPrepareDC() right before it displays data onscreen or before it prints the data to the printer. (One strength of the device context approach to screen display is that the same code can often be used for display and printing.) If the application is about to display data, you (probably) don't want to change the default processing performed by OnPrepareDC(). So, you must check whether the application is printing data by calling IsPrinting(), a member function of the device context class.

If the application is printing, you must determine which part of the data belongs on the current page. You need the height in dots of a printed page, so you call GetDeviceCaps() again. Next, you must determine a new viewport origin (the position of the coordinates 0,0) for the display. Changing the origin tells MFC where to begin displaying data. For page one, the origin is zero; for page two, it's moved down by the number of dots on a page. In general, the vertical component is the page size times the

current page minus one. The page number is a member variable of the CPrintInfo class.

After you calculate the new origin, you only need to give it to the device context by calling SetViewportOrg(). Your changes to OnPrepareDC() are complete. To see your changes in action, build and run your new version of Print1. When the program's main window appears, click twice in the window to add two rectangles to the display. (The displayed rectangle count should be seven.) Again, choose File, Print Preview and look at the two-page print preview window (see Figure 6.14). Now the program previews the document correctly. If you print the document, it will look the same in hard copy as it does in the preview.

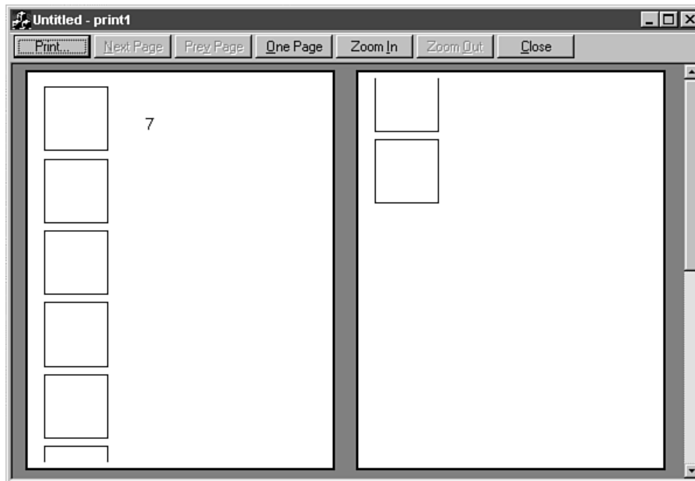


FIG. 6.14 Print1 finally previews and prints properly.

MFC and Printing

Now you've seen MFC's printing and print preview support in action. As you added more functionality to the Print1 application, you modified several member functions that were overridden in the view class, including OnDraw(), OnBeginPrinting(), and OnPrepareDC(). These functions are important to the printing and print preview processes. However, other functions also enable you to add even more printing power to your applications. Table 6.2 describes the functions important to the printing process.

Table 6.2 Printing Functions of a View Class

Function	Description
OnBeginPrinting()	Override this function to create resources, such as fonts, that you need for printing the document. You also set the maximum page count here.
OnDraw()	This function serves triple duty, displaying data in a frame

	window, a print preview window, or on the printer, depending on the device context sent as the function's parameter.
OnEndPrinting()	Override this function to release resources created in OnBeginPrinting().
OnPrepareDC()	Override this function to modify the device context used to display or print the document. You can, for example, handle pagination here.
OnPreparePrinting()	Override this function to provide a maximum page count for the document. If you don't set the page count here, you should set it in OnBeginPrinting().
OnPrint()	Override this function to provide additional printing services, such as printing headers and footers, not provided in OnDraw().

To print a document, MFC calls the functions listed in Table 6.2 in a specific order. First it calls OnPreparePrinting(), which simply calls DoPreparePrinting(), as shown in Listing 6.6. DoPreparePrinting() is responsible for displaying the Print dialog box and creating the printer DC.

Listing 6.6 print1View.cpp —CPrint1View::OnPreparePrinting() as Generated by AppWizard

```
BOOL CPrint1View::OnPreparePrinting(CPrintInfo* pInfo)

{

    // default preparation

    return DoPreparePrinting(pInfo);

}
```

As you can see, OnPreparePrinting() receives as a parameter a pointer to a CPrintInfo object. By using this object, you can obtain information about the print job as well as initialize attributes such as the maximum page number. Table 6.3 describes the most useful data and function members of the CPrintInfo class.

Table 6.3 Members of the CPrintInfo Class

Member	**Description**
SetMaxPage()	Sets the document's maximum page number.
SetMinPage()	Sets the document's minimum page number.

GetFromPage() Gets the number of the first page that users selected for printing.

GetMaxPage() Gets the document's maximum page number, which may be changed in On Begin Printing ().

GetMinPage() Gets the document's minimum page number, which may be changed in OnBeginPrinting().

GetToPage() Gets the number of the last page users selected for printing.

m_bContinue Printing Controls the printing process. Setting the flag to FALSE ends the print job.

m_bDirect Indicates whether the document is being directly printed.

m_bPreview Indicates whether the document is in print preview.

m_nCurPage Holds the current number of the page being printed.

m_nNumPreviewPages Holds the number of pages (1 or 2) being displayed in print preview.

m_pPD Holds a pointer to the print job's CPrintDialog object.

m_rect Draw Holds a rectangle that defines the usable area for the current page.

m_str Page Desc Holds a page-number format string.

When the DoPreparePrinting() function displays the Print dialog box, users can set the value of many data members of the CPrintInfo class. Your program then can use or set any of these values. Usually, you'll at least call SetMaxPage(), which sets the document's maximum page number, before DoPreparePrinting() so that the maximum page number displays in the Print dialog box. If you can't determine the number of pages until you calculate a page length based on the selected printer, you have to wait until you have a printer DC for the printer. After OnPreparePrinting(), MFC calls OnBeginPrinting(), which is not only another place to set the maximum page count but also the place to create resources, such as fonts, that you need to complete the print job. OnPreparePrinting() receives as parameters a pointer to the printer DC and a pointer to the associated CPrintInfo object.

Next, MFC calls OnPrepareDC() for the first page in the document. This is the beginning of a print loop that's executed once for each page in the document. OnPrepareDC() is the place to control what part of the whole document prints on the current page. As you saw previously, you handle this task by setting the document's viewport origin.

After OnPrepareDC(), MFC calls OnPrint() to print the actual page. Normally, OnPrint() calls OnDraw() with the printer DC, which automatically directs OnDraw()'s output to the printer rather than onscreen. You can override OnPrint() to control how the document is printed. You can print headers and footers in OnPrint() and then call

the base class's version (which in turn calls OnDraw()) to print the body of the document, as demonstrated in Listing 6.7. (The footer will appear below the body, even though PrintFooter() is called before OnPrint()—don't worry.) To prevent the base class version from overwriting your header and footer area, restrict the printable area by setting the m_rectDraw member of the CPrintInfo object to a rectangle that doesn't overlap the header or footer.

Listing 6.7 Possible OnPrint() with Headers and Footers

```
void CPrint1View::OnPrint(CDC* pDC, CPrintInfo* pInfo)

{

    // TODO: Add your specialized code here and/or call the base class

    // Call local functions to print a header and footer.

    PrintHeader();

    PrintFooter();

    CView::OnPrint(pDC, pInfo);

}
```

Alternatively, you can remove OnDraw() from the print loop entirely by doing your own printing in OnPrint() and not calling OnDraw() at all (see Listing 6.8).

Listing 6.8 Possible OnPrint() Without OnDraw()

```
void CPrint1View::OnPrint(CDC* pDC, CPrintInfo* pInfo)

{

    // TODO: Add your specialized code here and/or call the base class


    // Call local functions to print a header and footer.

    PrintHeader();

    PrintFooter();

    // Call a local function to print the body of the document.

    PrintDocument();

}
```

As long as there are more pages to print, MFC continues to call OnPrepareDC() and OnPrint() for each page in the document. After the last page is printed, MFC calls OnEndPrinting(), where you can destroy any resources you created in OnBeginPrinting(). Figure 6.15 summarizes the entire printing process.

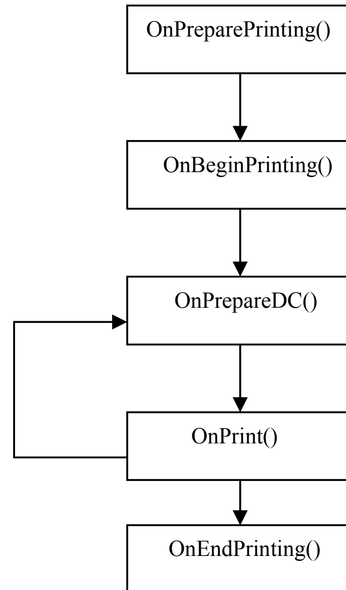


FIG. 6.15 MFC calls various member functions during the printing process.

UNIT – III

PERSISTENCE AND FILE I/O

In this chapter

Understanding Objects and Persistence

Examining the File Demo Application

Creating a Persistent Class

Reading and Writing Files Directly

Creating Your Own *CArchive* Objects

Using the Registry

Understanding Objects and Persistence

One of the most important things a program must do is save users' data after that data is changed in some way. Without the capability to save edited data, the work a user performs with an application exists only as long as the application is running, vanishing the instant the user exits the application. Not a good way to get work done! In many cases, especially when using AppWizard to create an application, Visual C++ provides much of the code necessary to save and load data. However, in some cases—most notably when you create your own object types—you have to do a little extra work to keep your users' files up to date. When you're writing an application, you deal with a lot of different object types. Some data objects might be simple types, such as integers and characters. Other objects might be instances of classes, such as strings from the CString class or even objects created from your own custom classes. When using objects in applications that must create, save, and load documents, you need a way to save and load the state of those objects so that you can re-create them exactly as users left them at the end of the last session. An object's capability to save and load its state is called *persistence*. Almost all MFC classes are persistent because they're derived directly or indirectly from MFC's CObject class, which provides the basic functionality for saving and loading an object's state. The following section reviews how MFC makes a document object persistent.

Examining the File Demo Application

When you use Visual C++'s AppWizard to create a program, you get an application that uses document and view classes to organize, edit, and display its data. As discussed in Chapter 4, "Documents and Views," the document object, derived from the CDocument class, is responsible for holding the application's data

during a session and for saving and loading the data so that the document persists from one session to another.

In this chapter, you'll build the File Demo application, which demonstrates the basic techniques behind saving and loading data of an object derived from CDocument. File Demo's document is a single string containing a short message, which the view displays.

Three menu items are relevant in the File Demo application. When the program first begins, the message is automatically set to the string Default Message. Users will change this message by choosing Edit, Change Message. The File, Save menu option saves the document, as you'd expect, and File, Open reloads it from disk.

A Review of Document Classes

Anyone who's written a program has experienced saving and opening files—object persistence from the user's point of view. In this chapter you'll learn how persistence works. Although you had some experience with document classes in Chapter 4, you'll now review the basic concepts with an eye toward extending those concepts to your own custom classes. When working with an application created by AppWizard, you must complete several steps to enable your document to save and load its state. Those steps are discussed in this section.

The steps are as follows:

1. Define the member variables that will hold the document's data.
2. Initialize the member variables in the document class's OnNewDocument() member function.
3. Display the current document in the view class's OnDraw() member function.
4. Provide member functions in the view class that enable users to edit the document.
5. Add to the document class's Serialize() member function the code needed to save and load the data that comprises the document.

When your application can handle multiple documents, you need to do a little extra work to be sure that you use, change, or save the correct document. Luckily, most of that work is taken care of by MFC and AppWizard.

Building the File Demo Application

To build the File Demo application, start by using AppWizard to create an SDI application. All the other AppWizard choices should be left at their default values, so you can speed things up by clicking Finish on Step 1 after selecting SDI and making sure that Document/View support is selected.

Double-click CfileDemoDoc in ClassView to edit the header file for the document class. In the Attributes section add a CString member variable called m_message, so that the Attributes section looks like this:

```
// Attributes

public:

CString m_message;
```

In this case, the document's storage is nothing more than a single string object. Usually, your document's storage needs are much more complex. This single string, however, is enough to demonstrate the basics of a persistent document. It's very common for MFC programmers to use public variables in their documents, rather than a private variable with public access functions. It makes it a little simpler to write the code in the view class that will access the document variables. It will, however, make future enhancements a little more work. These tradeoffs are discussed in more detail in Appendix A, "C++ Review and Object-Oriented Concepts."

This string, like all the document's data, must be initialized. The OnNewDocument() member function is the place to do it. Expand CFileDemoDoc in ClassView and double-click OnNewDocument() to edit it. Add a line of code to initialize the string so that the function looks like Listing 7.1. You should remove the TODO comments because you've done what they were reminding you to do.

Listing 7.1 Initializing the Document's Data

```
BOOL CFileDemoDoc::OnNewDocument()

{

    if (!CDocument::OnNewDocument())

    return FALSE;

    m_message = "Default Message";

    return TRUE;

}
```

With the document class's m_message data member initialized, the application can display the data in the view window. You just need to edit the view class's OnDraw() function (see Listing 7.2). Expand CFileDemoView in ClassView and double-click OnDraw() to edit it. Again, you're just adding one line of code and removing the TODO comment.

Listing 7.2 Displaying the Document's Data

```
void CFileDemoView::OnDraw(CDC* pDC)

{
```

```
    CFileDoc* pDoc = GetDocument();

    ASSERT_VALID(pDoc);

    pDC->TextOut(20, 20, pDoc->m_message);

}
```

Getting information onscreen, using device contexts, and the TextOut() function are all discussed in Chapter 5, “Drawing on the Screen.”

Build File Demo now, to make sure there are no typos, and run it. You should see Default Message appear onscreen.

Now, you need to allow users to edit the application’s document by changing the string. In theory, the application should display a dialog box to let the user enter any desired string at all. For our purposes, you’re just going to have the Edit, Change Message menu option assign the string a different, hard-coded value. ShowString, the subject of Chapter 8, “Building a Complete Application: ShowString,” shows how to create a dialog box such as the one File Demo might use.

Click the Resource tab to switch to ResourceView, expand the resources, expand Menus, and double-click IDR_MAINFRAME to edit it. Click once on the Edit item in the menu you are editing to drop it down. Click the blank item at the end of the list and type **Change &Message**. This will add another item to the menu.

Choose View, ClassWizard to make the connection between this menu item and your code. You should see ID_EDIT_CHANGEMESSAGE highlighted already; if not, click it in the box on the left to highlight it. Choose CFileDemoView from the drop-down box on the upper right. Click COMMAND in the lower-right box and then click the Add Function button. Accept the suggested name, OnEditChangemessage(), by clicking OK on the dialog that appears. Click Edit Code to open the new function in the editor and edit it to match Listing 7.3.

Listing 7.3 Changing the Document’s Data

```
void CFileDemoView::OnEditChangemessage()

{

    CTime now = CTime::GetCurrentTime();

    CString changetime = now.Format(“Changed at %B %d %H:%M:%S”);

    GetDocument()->m_message = changetime;

    GetDocument()->SetModifiedFlag();

    Invalidate();

}
```

This function, which responds to the application's Edit, Change Message command, builds a string from the current date and time and transfers it to the document's data member. (The CTime class and its Format() function are discussed in Appendix F, "Useful Classes.") The call to the document class's SetModifiedFlag() function notifies the object that its contents have been changed. The application will warn about exiting with unsaved changes as long as you remember to call SetModifiedFlag() everywhere there might be a change to the data. Finally, this code forces a redraw of the screen by calling Invalidate(), as discussed in Chapter 4.

The document class's Serialize() function handles the saving and loading of the document's data. Listing 7.4 shows the empty shell of Serialize() generated by AppWizard.

Listing 7.4 FILEVIEW.CPP—The Document Class Serialize() Function

```
void CFileDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
}
```

Because the CString class (of which m_message is an object) defines the >> and << operators for transferring strings to and from an archive, it's a simple task to save and load the document class's data. Simply add this line where the comment reminds you to add storing code:

```
ar << m_message;
```

Add this similar line where the loading code belongs:

```
ar >> m_message;
```

The << operator sends the CString m_message to the archive; the >> operator fills m_message from the archive. As long as all the document's member variables are simple data types such as integers or characters, or MFC classes such as CString with

these operators already defined, it's easy to save and load the data. The operators are defined for these simple data types:

- BYTE
- WORD
- Int
- LONG
- DWORD
- Float
- Double

Build File Demo and run it. Choose Edit, Change Message, and you should see the new string onscreen, as shown in Figure 7.1. Choose File, Save and enter a filename you can remember. Now change the message again. Choose File, New and you'll be warned about saving your current changes first, as in Figure 7.2. Choose File, Open and browse to your file, or just find your filename towards the bottom of the File menu to re-open it, and you'll see that File Demo can indeed save and reload a string.

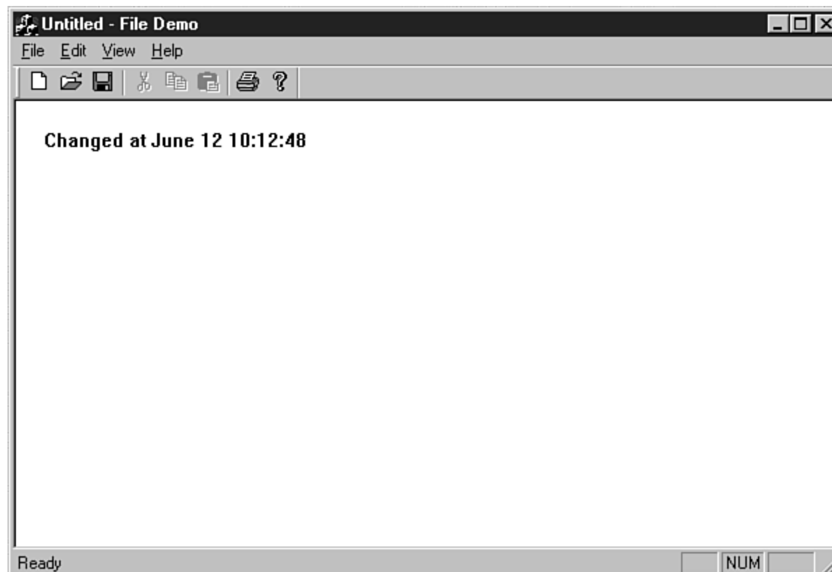


FIG. 7.1 File Demo changes the string on command.

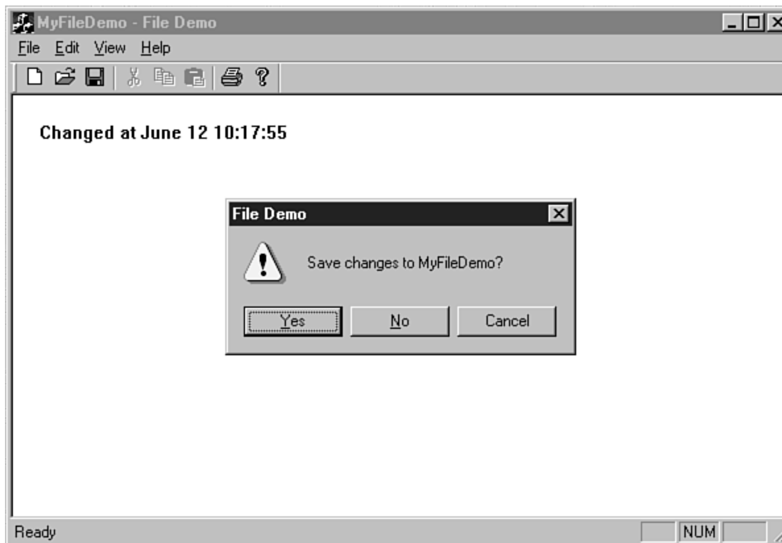


FIG. 7.2 Your users will never lose unsaved data again.

Creating a Persistent Class

What if you've created your own custom class for holding the elements of a document? How can you make an object of this class persistent? You find the answers to these questions in this section.

Suppose that you now want to enhance the File Demo application so that it contains its data in a custom class called CMessages. The member variable is now called m_messages and is an instance of CMessages. This class holds three CString objects, each of which must be saved and loaded for the application to work correctly. One way to arrange this is to save and load each individual string, as shown in Listing 7.5.

Listing 7.5 One Possible Way to Save the New Class's Strings

```
void CFileDoc::Serialize(CArchive& ar)

{

    if (ar.IsStoring())

    {

        ar << m_messages.m_message1;

        ar << m_messages.m_message2;

        ar << m_messages.m_message3;

    }
```

```
        else
        {
                ar >> m_messages.m_message1;
                ar >> m_messages.m_message2;
                ar >> m_messages.m_message3;
        }
    }
```

You can write the code in Listing 7.5 only if the three member variables of the CMessages class are public and if you know the implementation of the class itself. Later, if the class is changed in any way, this code also has to be changed. It's more object oriented to delegate the work of storing and loading to the CMessages class itself. This requires some preparation. The following basic steps create a class that can serialize its member variables:

1. Derive the class from CObject.

2. Place the DECLARE_SERIAL() macro in the class declaration.

3. Place the IMPLEMENT_SERIAL() macro in the class implementation.

4. Override the Serialize() function in the class.

5. Provide an empty, default constructor for the class.

In the following section, you build an application that creates persistent objects in just this way.

The File Demo 2 Application

The next sample application, File Demo 2, demonstrates the steps you take to create a class from which you can create persistent objects. It will have an Edit, Change Messages command that changes all three strings. Like File Demo, it will save and reload the document when the user chooses File, Save or File, Open.

Build an SDI application called MultiString just as you built File Demo. Add a member variable to the document, as before, so that the Attributes section of MultiStringDoc.h reads

```
// Attributes
public:
    CMessages m_messages;
```

The next step is to write the CMessages class.

Looking at the *CMessages* Class

Before you can understand how the document class manages to save and load its contents successfully, you have to understand how the CMessages class, of which the document class's m_messages data member is an object, works. As you work with this class, you will see how to implement the preceding five steps for creating a persistent class.

To create the CMessages class, first choose Insert, New Class. Change the class type to generic class and name it CMessages. In the area at the bottom of the screen, enter CObject as the base class name and leave the As column set to public, as shown in Figure 7.3.

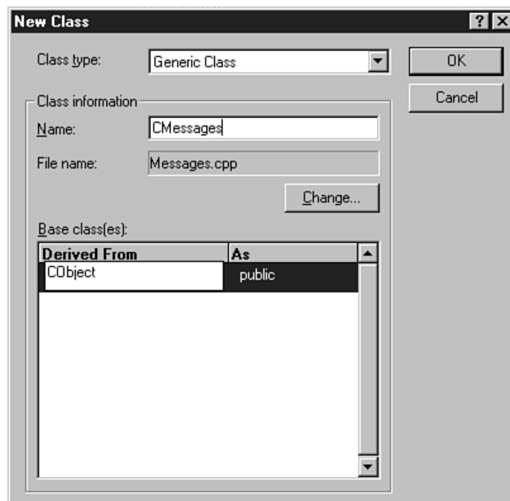


FIG. 7.3 Create a new class to hold the messages.

This will create two files: messages.h for the header and messages.cpp for the code. It also adds some very simple code to these files for you. (You may get a warning about not being able to find the header file for CObject: just click OK and ignore it because CObject, like all MFC files, is available to you without including extra headers.) Switch back to Multistringdoc.h and add this line before the class definition:

```
#include "Messages.h"
```

This will ensure the compiler knows about the CMessages class when it compiles the document class. You can build the project now if you want to be sure you haven't forgotten anything. Now switch back to Messages.h and add these lines:

```
    DECLARE_SERIAL(CMessages)

protected:

    CString m_message1;

    CString m_message2;
```

```
    CString m_message3;
public:
    void SetMessage(UINT msgNum, CString msg);
    CString GetMessage(UINT msgNum);
    void Serialize(CArchive& ar);
```

The DECLARE_SERIAL() macro provides the additional function and member variable declarations needed to implement object persistence.

Next come the class's data members, which are three objects of the CString class. Notice that they are protected member variables. The public member functions are next. SetMessage(), whose arguments are the index of the string to set and the string's new value, changes a data member. GetMessage() is the complementary function, enabling a program to retrieve the current value of any of the strings. Its single argument is the number of the string to retrieve.

Finally, the class overrides the Serialize() function, where all the data saving and loading takes place. The Serialize() function is the heart of a persistent object, with each persistent class implementing it in a different way. Listing 7.6 shows the code for each of these new member functions. Add it to messages.cpp.

Listing 7.6 MESSAGES.CPP—The CMessages Class Implementation File

```
void CMessages::SetMessage(UINT msgNum, CString msg)
{
    switch (msgNum)
    {
        case 1:
            m_message1 = msg;
            break;
        case 2:
            m_message2 = msg;
            break;
        case 3:
            m_message3 = msg;
            break;
    }
    SetModifiedFlag();
}
```

```
CString CMessages::GetMessage(UINT msgNum)
{
    switch (msgNum)
    {
        case 1:
        return m_message1;
    case 2:
        return m_message2;
    case 3:
        return m_message3;
    default:
        return "";
    }
}
void CMessages::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);
    if (ar.IsStoring())
    {
        ar << m_message1 << m_message2 << m_message3;
    }
    else
    {
        ar >> m_message1 >> m_message2 >> m_message3;
    }
}
```

There's nothing tricky about the SetMessage() and GetMessage() functions, which perform their assigned tasks precisely. The Serialize() function, however, may inspire a couple of questions. First, note that the first line of the body of the function calls the base class's Serialize() function. This is a standard practice for many functions that override functions of a base class. In this case, the call to CObject::Serialize() doesn't do much because the CObject class's Serialize() function is empty. Still, calling the base class's Serialize() function is a good habit to get into because you may not always be working with classes derived directly from CObject.

After calling the base class's version of the function, Serialize() saves and loads its data in much the same way a document object does. Because the data members

that must be serialized are CString objects, the program can use the >> and << operators to write the strings to the disk.

Towards the top of messages.cpp, after the include statements, add this line:

```
IMPLEMENT_SERIAL(CMessages, CObject, 0)
```

The IMPLEMENT_SERIAL() macro is partner to the DECLARE_SERIAL() macro,

providing implementation for the functions that give the class its persistent capabilities. The macro's three arguments are the name of the class, the name of the immediate base class, and a schema number, which is like a version number. In most cases, you use 0 or 1 for the schema number.

Using the *CMessages* Class in the Program

Now that CMessages is defined and implemented, member functions of the MultiString document and view classes can work with it. First, expand CMultiStringDoc and double-click OnNewDocument() to edit it. Add these lines in place of the TODO comments.

```
m_messages.SetMessage(1, "Default Message 1");

m_messages.SetMessage(2, "Default Message 2");

m_messages.SetMessage(3, "Default Message 3");
```

Because the document class can't directly access the data object's protected data members, it initializes each string by calling the CMessages class's SetMessage() member function. Expand CMultiStringView and double-click OnDraw() to edit it. Here's how it should look when you're finished:

```
void CMultiStringView::OnDraw(CDC* pDC)

{

    CMultiStringDoc* pDoc = GetDocument();

    ASSERT_VALID(pDoc);

    pDC->TextOut(20, 20, pDoc->m_messages.GetMessage(1));

    pDC->TextOut(20, 40, pDoc->m_messages.GetMessage(2));

    pDC->TextOut(20, 60, pDoc->m_messages.GetMessage(3));

}
```

As you did for File Demo, add a “Change Messages” item to the Edit menu. Connect it to a view function called OnEditChangemessages. This function will change the data by calling the CMessages object’s member functions, as in Listing 7.7. The view class’s OnDraw() function also calls the GetMessage() member function to access the CMessages class’s strings.

Listing 7.7 Editing the Data Strings

```
void CMultiStringView::OnEditChangemessages()
{
    CMultiStringDoc* pDoc = GetDocument();
    CTime now = CTime::GetCurrentTime();
    CString changetime = now.Format(“Changed at %B %d %H:%M:%S”);
    pDoc->m_messages.SetMessage(1, CString(“String 1 “) + changetime);
    pDoc->m_messages.SetMessage(2, CString(“String 2 “) + changetime);
    pDoc->m_messages.SetMessage(3, CString(“String 3 “) + changetime);
    pDoc->SetModifiedFlag();
    Invalidate();
}
```

All that remains is to write the document class’s Serialize() function, where the m_messages data object is serialized out to disk. You just delegate the work to the data object’s own Serialize() function, as in Listing 7.8.

Listing 7.8 Serializing the Data Object

```
void CMultiStringDoc::Serialize(CArchive& ar)
{
    m_messages.Serialize(ar);
    if (ar.IsStoring())
    {
    }
    else
    {
```

```
    }
}
```

As you can see, after serializing the m_messages data object, not much is left to do in the document class's Serialize() function. Notice that the call to m_messages.Serialize() passes the archive object as its single parameter. Build MultiString now and test it as you tested File Demo. It should do everything you expect.

Reading and Writing Files Directly

Although using MFC's built-in serialization capabilities is a handy way to save and load data, sometimes you need more control over the file-handling process. For example, you might need to deal with your files nonsequentially, something the Serialize() function and its associated CArchive object can't handle because they do stream I/O. In this case, you can handle files almost exactly as they're handled by non-Windows programmers: creating, reading, and writing files directly. Even when you need to dig down to this level of file handling, MFC offers help. Specifically, you can use the CFile class and its derived classes to handle files directly.

The *CFile* Class

MFC's CFile class encapsulates all the functions you need to handle any type of file. Whether you want to perform common sequential data saving and loading or construct a random access file, the CFile class gets you there. Using the CFile class is a lot like handling files the oldfashioned C-style way, except that the class hides some of the busy-work details from you so that you can get the job done quickly and easily. For example, you can create a file for reading with only a single line of code. Table 7.1 shows the CFile class's member functions and their descriptions.

Table 7.1 Member Functions of the CFile Class

Function	Description
CFile	Creates the CFile object. If passed a filename, it opens the file.
Destructor	Cleans up a CFile object that's going out of scope. If the file is open, it closes that file.
Abort()	Immediately closes the file with no regard for errors.
Close()	Closes the file.
Duplicate()	Creates a duplicate file object.

Flush()	Flushes data from the stream.
GetFileName()	Gets the file's filename.
GetFilePath()	Gets the file's full path.
GetFileTitle()	Gets the file's title (the filename without the extension).
Function	**Description**
GetLength()	Gets the file's length.
GetPosition()	Gets the current position within the file.
GetStatus()	Gets the file's status.
LockRange()	Locks a portion of the file.
Open()	Opens the file.
Read()	Reads data from the file.
Remove()	Deletes a file.
Rename()	Renames the file.
Seek()	Sets the position within the file.
SeekToBegin()	Sets the position to the beginning of the file.
SeekToEnd()	Sets the position to the end of the file.
SetFilePath()	Sets the file's path.
SetLength()	Sets the file's length.
SetStatus()	Sets the file's status.
UnlockRange()	Unlocks a portion of the file.
Write()	Writes data to the file.

As you can see from Table 7.1, the CFile class offers plenty of file-handling power. This section demonstrates how to call a few of the CFile class's member functions. However, most of the other functions are just as easy to use. Here's a sample snippet of code that creates and opens a file, writes a string to it, and then gathers some information about the file:

```
// Create the file.

CFile file("TESTFILE.TXT", CFile::modeCreate | CFile::modeWrite);
```

```
// Write data to the file.

CString message("Hello file!");

int length = message.GetLength();

file.Write((LPCTSTR)message, length);

// Obtain information about the file.

CString filePath = file.GetFilePath();

Int fileLength = file.GetLength();
```

Notice that you don't have to explicitly open the file when you pass a filename to the constructor, whose arguments are the name of the file and the file access mode flags. You can use several flags at a time simply by ORing their values together, as in the little snippet above. These flags, which describe how to open the file and which specify the types of valid operations, are defined as part of the CFile class and are described in Table 7.2.

Table 7.2 The File Mode Flags

Flag	Description
CFile::modeCreate	Creates a new file or truncates an existing file to length 0
CFile::modeNoInherit	Disallows inheritance by a child process
CFile::modeNoTruncate	When creating the file, doesn't truncate the file if it already exists
CFile::modeRead	Allows read operations only
CFile::modeReadWrite	Allows both read and write operations
CFile::modeWrite	Allows write operations only
CFile::shareCompat	Allows other processes to open the file
CFile::shareDenyNone	Allows other processes read or write operations on the file
CFile::shareDenyRead	Disallows read operations by other processes

CFile::shareDenyWrite	Disallows write operations by other processes
CFile::shareExclusive	Denies all access to other processes
CFile::typeBinary	Sets binary mode for the file
CFile::typeText	Sets text mode for the file

CFile::Write() takes a pointer to the buffer containing the data to write and the number of bytes to write. Notice the LPCTSTR casting operator in the call to Write(). This operator is defined by the CString class and extracts the string from the class.

One other thing about the code snippet: There is no call to Close()—the CFile destructor closes the file automatically when file goes out of scope. Reading from a file isn't much different from writing to one:

```
// Open the file.

CFile file("TESTFILE.TXT", CFile::modeRead);

// Read data from the file.

char s[81];

int bytesRead = file.Read(s, 80);

s[bytesRead] = 0;

CString message = s;
```

This time the file is opened by the CFile::modeRead flag, which opens the file for read operations only, after which the code creates a character buffer and calls the file object's Read() member function to read data into the buffer. The Read() function's two arguments are the buffer's address and the number of bytes to read. The function returns the number of bytes actually read, which in this case is almost always less than the 80 requested. By using the number of bytes read, the program can add a 0 to the end of the character data, thus creating a standard C-style string that can be used to set a CString variable.

The code snippets you've just seen use a hard-coded filename. To get filenames from your user with little effort, be sure to look up the MFC class CFileDialog in the online help. It's simple to use and adds a very nice touch to your programs.

Creating Your Own *CArchive* Objects

Although you can use CFile objects to read from and write to files, you can also go a step farther and create your own CArchive object and use it exactly as you use the CArchive object in the Serialize() function. This lets you take advantage of Serialize

functions already written for other objects, passing them a reference to your own archive object.

To create an archive, create a CFile object and pass it to the CArchive constructor. For example, if you plan to write out objects to a file through an archive, create the archive like this:

```
CFile file("FILENAME.EXT", CFile::modeWrite);

CArchive ar(&file, CArchive::store);
```

After creating the archive object, you can use it just like the archive objects that MFC creates for you, for example, calling Serialize() yourself and passing the archive to it. Because you created the archive with the CArchive::store flag, any calls to IsStoring() return TRUE, and the code that dumps objects to the archive executes. When you're through with the archive object, you can close the archive and the file like this:

```
ar.Close();

file.Close();
```

If the objects go out of scope soon after you're finished with them, you can safely omit the calls to Close() because both CArchive and CFile have Close() calls in the destructor.

Using the Registry

In the early days of Windows programming, applications saved settings and options in initialization files, typically with the .INI extension. The days of huge WIN.INI files or myriad private .INI files are now gone—when an application wants to store information about itself, it does so by using a centralized system Registry. Although the Registry makes sharing information between processes easier, it can make things more confusing for programmers. In this section, you uncover some of the mysteries of the Registry and learn how to manage it in your applications.

How the Registry Is Set Up

Unlike .INI files, which are plain text files that can be edited with any text editor, the Registry contains binary and ASCII information that can be edited only by using the Registry Editor or special API function calls created specifically for managing the Registry. If you've ever used the Registry Editor to browse your system's Registry, you know that it contains a huge amount of information that's organized into a tree structure. Figure 7.4 shows how the Registry appears when you first run the Registry Editor. (On Windows 95, you can find the Registry Editor, REGEDIT.EXE, in your main Windows folder, or you can run it from the Start menu by choosing Run, typing **regedit**, and then clicking OK. Under Windows NT, it's REGEDT32.EXE.)

The far left window lists the Registry's predefined keys. The plus marks next to the keys in the tree indicate that you can open the keys and view more detailed

information associated with them. Keys can have subkeys, and subkeys themselves can have subkeys. Any key or subkey may or may not have a value associated with it. If you explore deep enough in the hierarchy, you see a list of values in the far right window. In Figure 7.5, you can see the values associated with the current user's screen appearance. To see these values yourself, browse from HKEY_CURRENT_USER to Control Panel to Appearance to Schemes, and you'll see the desktop schemes installed on your system.

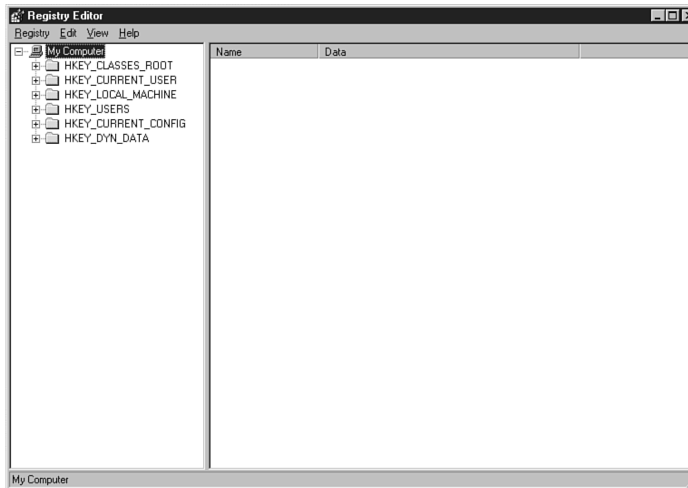


FIG. 7.4 The Registry Editor displays the Registry.

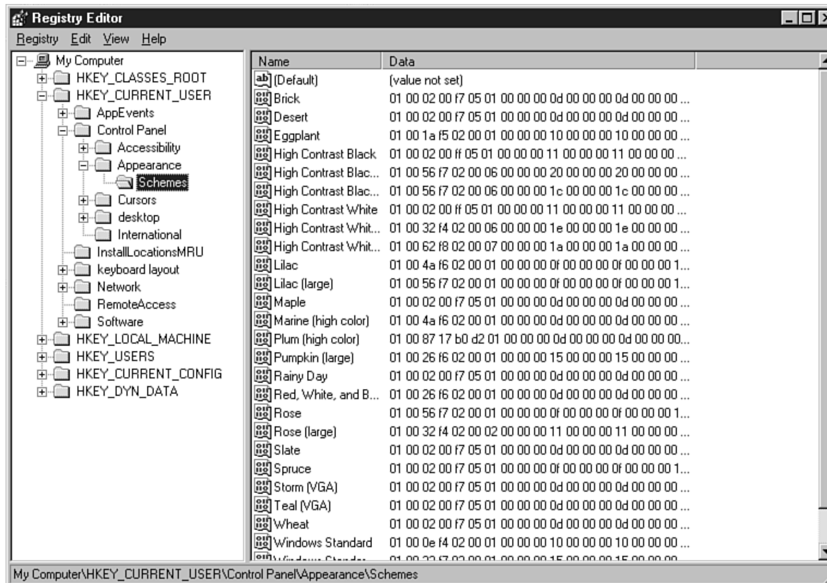


FIG. 7.5 The Registry is structured as a tree containing a huge amount of information.

The Predefined Keys

To know where things are stored in the Registry, you need to know about the predefined keys and what they mean. From Figure 7.4, you can see that the six predefined keys are

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS
- HKEY_CURRENT_CONFIG
- HKEY_DYN_DATA

The HKEY_CLASSES_ROOT key holds document types and properties, as well as class information about the various applications installed on the machine. For example, if you explored this key on your system, you'd probably find an entry for the .DOC file extension, under which you'd find entries for the applications that can handle this type of document (see Figure 7.6).

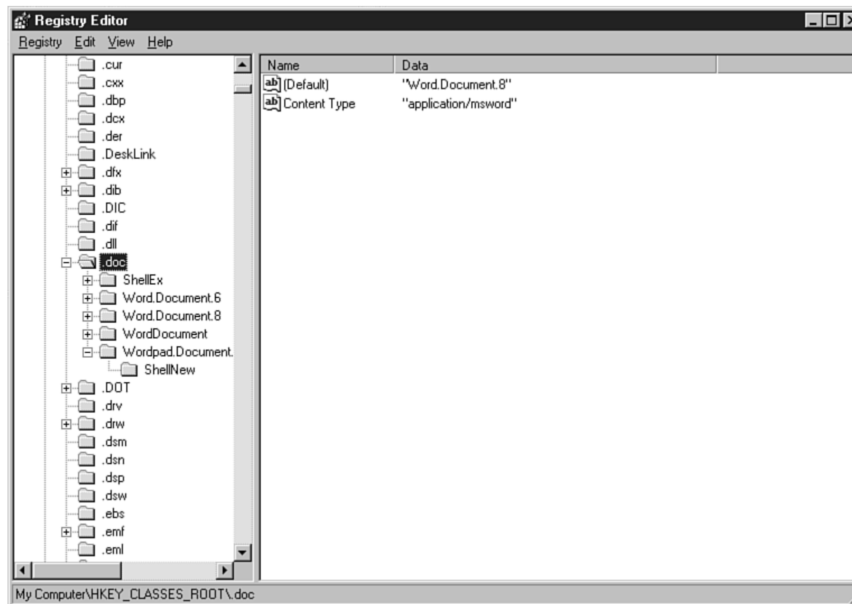


FIG. 7.6 The HKEY_CLASSES_ROOT key holds document information.

The HKEY_CURRENT_USER key contains all the system settings the current user has established, including color schemes, printers, and program groups. The HKEY_LOCAL_MACHINE key, on the other hand, contains status information about the computer, and the HKEY_USERS key organizes information about each user of the system, as well as the default configuration. Finally, the HKEY_CURRENT_CONFIG

key holds information about the hardware configuration, and the HKEY_DYN_DATA key contains information about dynamic Registry data, which is data that changes frequently. (You may not always see this key on your system.)

Using the Registry in an MFC Application

Now that you know a little about the Registry, let me say that it would take an entire book to explain how to fully access and use it. As you may imagine, the Win32 API features many functions for manipulating the Registry. If you're going to use those functions, you had better know what you're doing! Invalid Registry settings can crash your machine, make it unbootable, and perhaps force you to reinstall Windows to recover.

However, you can easily use the Registry with your MFC applications to store information that the application needs from one session to another. To make this task as easy as possible, MFC provides the CWinApp class with the SetRegistryKey() member function, which creates (or opens) a key entry in the Registry for your application. All you have to do is supply a key name (usually a company name) for the function to use, like this:

SetRegistryKey("MyCoolCompany");

You should call SetRegistryKey() in the application class's InitInstance() member function, which is called once at program startup. After you call SetRegistryKey(), your application can create the subkeys and values it needs by calling one of two functions. The WriteProfileString() function adds string values to the Registry, and the WriteProfileInt() function adds integer values to the Registry. To get values from the Registry, you can use the GetProfileString() and GetProfileInt() functions. (You also can use RegSetValueEx() and RegQueryValueEx() to set and retrieve Registry values.)

The Sample Applications Revisited

In this chapter, you've already built applications that used the Registry. Here's an excerpt from

CMultiStringApp::InitInstance()—this code was generated by AppWizard and is also in CFileDemoApp::InitInstance().

```
// Change the registry key under which our settings are stored.

// You should modify this string to be something appropriate

// such as the name of your company or organization.

SetRegistryKey(_T("Local AppWizard-Generated Applications"));

LoadStdProfileSettings(); // Load standard INI file options (including MRU)
```

MRU stands for *Most Recently Used* and refers to the list of files that appears on the File menu after you open files with an application. Figure 7.7 shows the Registry Editor displaying the key that stores this information, HKEY_CURRENT_USER\Software\LocalAppWizard-Generated Applications\MultiString\Recent File List. In the foreground, MultiString's File menu shows the single entry in the MRU list

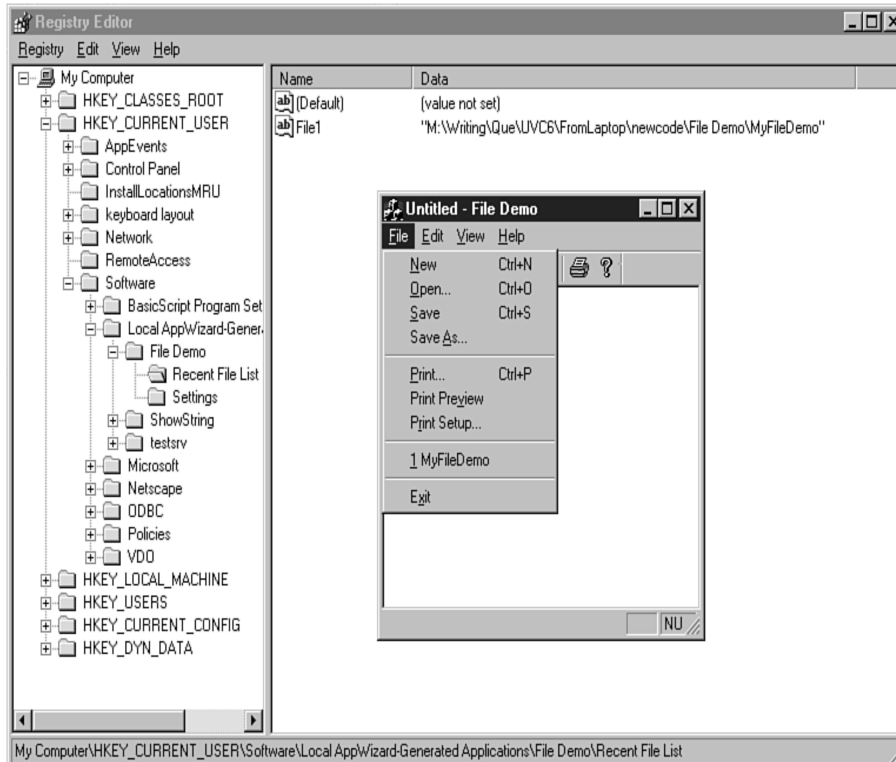


FIG. 7.7 The most recently used files list is stored in the Registry automatically.

BUILDING A COMPLETE APPLICATION: SHOWSTRING

In this chapter

Building an Application That Displays a String

Building the ShowString Menus

Building the ShowString Dialog Boxes

Making the Menu Work

Making the Dialog Box Work

Adding Appearance Options to the Options Dialog Box

Building an Application That Displays a String

In this chapter you pull together the concepts demonstrated in previous chapters to create an application that really does something. You add a menu, a menu item, a dialog box, and persistence to an application that draws output based on user settings. In subsequent chapters this application serves as a base for more advanced work.

The sample application you will build is very much like the traditional “Hello, world!” of C programming. It simply displays a text string in the main window. The *document* (what you save in a file) contains the string and a few settings. There is a new menu item to bring up a dialog box to change the string and the settings, which control the string’s appearance. This is a deliberately simple application so that the concepts of adding menu items and adding dialogs are not obscured by trying to understand the actual brains of the application. So, bring up Developer Studio and follow along.

Creating an Empty Shell with AppWizard

First, use AppWizard to create the starter application. (Chapter 1, “Building Your First Windows Application,” covers AppWizard and creating starter applications.) Choose File, New and the Project tab. Select an MFC AppWizard (exe) application, name the project ShowString so that your classnames will match those shown throughout this chapter, and click OK.

In Step 1 of AppWizard, it doesn't matter much whether you choose SDI or MDI, but MDI will enable you to see for yourself how little effort is required to have multiple documents open at once. So, choose MDI. Choose U.S. English, and then click Next.

The ShowString application needs no database support and no compound document support, so click Next on Step 2 and Step 3 without changing anything. In AppWizard's Step 4 dialog box, select a docking toolbar, initial status bar, printing and print preview, context-sensitive help, and 3D controls, and then click Next. Choose source file comments and shared DLL, and then click Next. The classnames and filenames are all fine, so click Finish. Figure 8.1 shows the final confirmation dialog box. Click OK.

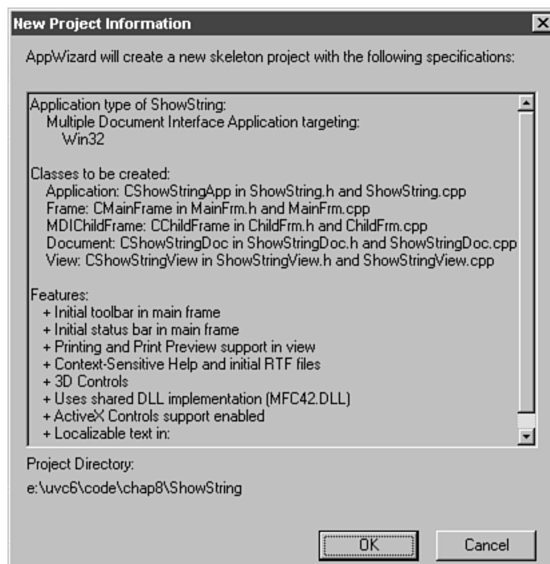


FIG. 8.1 AppWizard summarizes the design choices for ShowString.

Displaying a String

The ShowString application displays a string that will be kept in the document. You need to add a member variable to the document class, CShowStringDoc, and add loading and saving code to the Serialize() function. You can initialize the string by adding code to OnNewDocument() for the document and, in order to actually display it, override OnDraw() for the view. Documents and views are introduced in Chapter 4, "Documents and Views."

Member Variable and Serialization Add a private variable to the document and a public function to get the value by adding these lines to ShowStringDoc.h:

```
private:

        CString string;

public:
```

```
CString GetString() {return string;}
```

The inline function gives other parts of your application a copy of the string to use whenever necessary but makes it impossible for other parts to change the string.

Next, change the skeleton CShowStringDoc::Serialize() function provided by AppWizard to look like Listing 8.1. (Expand CShowStringDoc in ClassView and double-click Serialize() to edit the code.) Because you used the MFC CString class, the archive has operators << and >> already defined, so this is a simple function to write. It fills the archive from the string when you are saving the document and fills the string from the archive when you are loading the document from a file. Chapter 7, "Persistence and File I/O," introduces serialization.

Listing 8.1 SHOWSTRINGDOC.CPP—

CShowStringDoc::Serialize()

```
void CShowStringDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << string;
    }
    else
    {
        ar >> string;
    }
}
```

Initializing the String Whenever a new document is created, you want your application to initialize string to "Hello, world!". A new document is created when the user chooses File, New. This message is caught by CShowStringApp (the message map is shown in Listing 8.2, you can see it yourself by scrolling toward the top of ShowString.cpp) and handled by CWinApp::OnFileNew(). (Message maps and message handlers are discussed in Chapter 3, "Messages and Commands.") Starter applications generated by AppWizard call OnFileNew() to create a blank document when they run. OnFileNew() calls the document's OnNewDocument(), which actually initializes the member variables of the document.

Listing 8.2 SHOWSTRING.CPP—Message Map

```
BEGIN_MESSAGE_MAP(CShowStringApp, CWinApp)
    //{{AFX_MSG_MAP(CShowStringApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
        // NOTE - The ClassWizard will add and remove mapping macros here.
        // DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG_MAP
    // Standard file-based document commands
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    // Standard print setup command
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

AppWizard gives you the simple OnNewDocument() shown in Listing 8.3. To see yours in the editor, double-click OnNewDocument() in ClassView—you may have to expand CshowStringDoc first.

Listing 8.3 SHOWSTRINGDOC.CPP— CShowStringDoc::OnNewDocument()

```
BOOL CShowStringDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)

    return TRUE;
}
```

Take away the comments and add this line in their place:

```
string = “Hello, world!”;
```

(What else could it say, after all?) Leave the call to CDocument::OnNewDocument() because that will handle all other work involved in making a new document.

Getting the String Onscreen As you learned in Chapter 5, “Drawing on the Screen,” a view’s OnDraw() function is called whenever that view needs to be drawn, such as when your application is first started, resized, or restored or when a window that had been covering it is taken away. AppWizard has provided a skeleton, shown in Listing 8.4. To edit this function, expand CShowStringView in ClassView and then double-click OnDraw().

Listing 8.4 SHOWSTRINGVIEW.CPP—

CShowStringView::OnDraw()

```
void CShowStringView::OnDraw(CDC* pDC)
{
    CShowStringDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // TODO: add draw code for native data here
}
```

OnDraw() takes a pointer to a device context, as discussed in Chapter 5. The device context class, CDC, has a member function called DrawText() that draws text onscreen. It is declared like this:

```
int DrawText( const CString& str, LPRECT lpRect, UINT nFormat )
```

The CString to be passed to this function is going to be the string from the document class, which can be accessed as pDoc->GetString(). The lpRect is the client rectangle of the view, returned by GetClientRect(). Finally, nFormat is the way the string should display; for example, DT_CENTER means that the text should be centered from left to right within the view. DT_VCENTER means that the text should be centered up and down, but this works only for single lines of text that are identified with DT_SINGLELINE. Multiple format flags can be combined with |, so

DT_CENTER|DT_VCENTER|DT_SINGLELINE is the nFormat that you want. The drawing code to be added to CShowStringView::OnDraw() looks like this:

```
CRect rect;

GetClientRect(&rect);

pDC->DrawText(pDoc->GetString(), &rect,
DT_CENTER|DT_VCENTER|DT_SINGLELINE);
```

This sets up a CRect and passes its address to GetClientRect(), which sets the CRect to the client area of the view. DrawText() draws the document's string in the rectangle, centered vertically and horizontally. At this point, the application should display the string properly. Build and execute it, and you will see something like Figure 8.2. You have a lot of functionality—menus, toolbars, status bar, and so on—but nothing that any other Windows application doesn't have, yet. Starting with the next section, that changes.

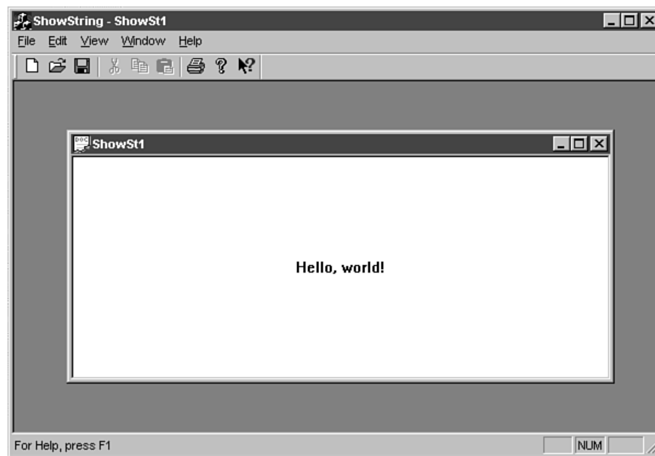


FIG. 8.2 ShowString starts simply, with the usual greeting.

Building the ShowString Menus

AppWizard creates two menus for you, shown in the ResourceView window in Figure 8.3. IDR_MAINFRAME is the menu shown when no file is open; IDR_SHOWSTTYPE is the menu shown when a ShowString document is open. Notice that IDR_MAINFRAME has no Window menus and that the File menu is much shorter than the one on the IDR_SHOWSTTYPE menu, with only New, Open, Print Setup, recent files, and Exit items.

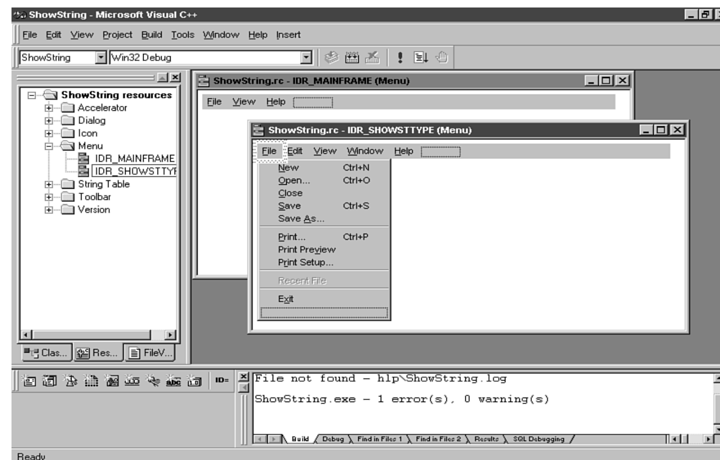


FIG. 8.3 AppWizard creates two menus for ShowString.

You are going to add a menu item to ShowString, so the first decision is where to add it. The user will be able to edit the string that displays and to set the string's format. You could add a Value item to the Edit menu that brings up a small dialog box for only the string and then create a Format menu with one item, Appearance, that brings up the dialog box to set the appearance. The choice you are going to see here, though, is to combine everything into one dialog box and then put it on a new Tools menu, under the Options item.

Do you need to add the item to both menus? No. When there is no document open, there is nowhere to save the changes made with this dialog box. So only IDR_SHOWSTTYPE needs to have a menu added. Open the menu by double-clicking it in the ResourceView window. At the far right of the menu, after Help, is an empty menu. Click it and type **&Tools**. The Properties dialog box appears; pin it to the background by clicking the pushpin. The Caption box contains &Tools. The menu at the end becomes the Tools menu, with an empty item underneath it; another empty menu then appears to the right of the Tools menu, as shown in Figure 8.4.

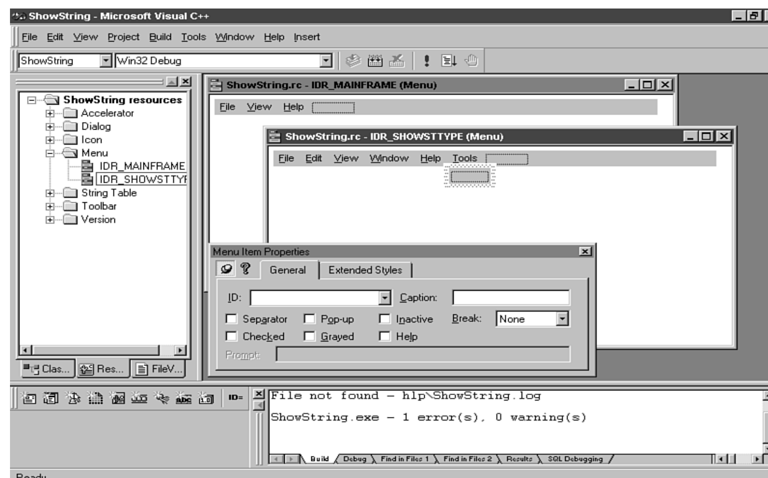


FIG. 8.4 Adding the Tools menu is easy in the ResourceView window.

Click the new Tools menu and drag it between the View and Window menus, corresponding to the position of Tools in products like Developer Studio and Microsoft Word. Next, click the empty sub-item. The Properties dialog box changes to show the blank properties of this item; change the caption to **&Options** and enter a sensible prompt, as shown in Figure 8.5. The prompt will be shown on the status bar when the user pauses the mouse over the menu item or moves the highlight over it with the cursor.

All menu items have a resource ID, and this resource ID is the way the menu items are connected to your code. Developer Studio will choose a good one for you, but it doesn't appear right away in the Properties dialog box. Click some other menu item, and then click Options again; you see that the resource ID is ID_TOOLS_OPTIONS. Alternatively, press Enter when you are finished, and the highlight moves down to the empty menu item below Options. Press the up-arrow cursor key to return the highlight to the Options item. If you'd like to provide an accelerator, like the Ctrl+C for Edit, Copy that the system provides, this is a good time to do it. Click the + next to Accelerator in the ResourceView window and then double-click IDR_MAINFRAME, the only Accelerator table in this application. At a glance, you can see what key combinations are already in use. Ctrl+O is already taken, but Ctrl+T is available. To connect Ctrl+T to Tools, Options, follow these steps:

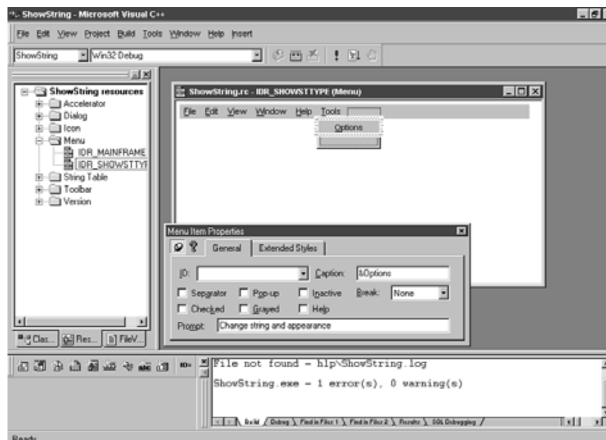


FIG. 8.5 The menu command Tools, Options controls everything that ShowString does.

1. Click the empty line at the bottom of the Accelerator table. If you have closed the Properties dialog box, bring it back by choosing View, Properties and then pin it in place. (Alternatively, double-click the empty line to bring up the Properties dialog box.)

2. Click the drop-down list box labeled ID and choose ID_TOOLS_OPTIONS from the list, which is in alphabetical order. (There are a lot of entries before ID_TOOLS_OPTIONS; drag the elevator down to almost the bottom of the list or start typing the resource ID—by the time you type ID_TO, the highlight will be in the right place.)

3. Type **T** in the Key box; then make sure that the Ctrl check box is selected and that the Alt and Shift boxes are deselected. Alternatively, click the Next Key Typed button and then type **Ctrl+T**, and the dialog box will be filled in properly.

4. Click another line in the Accelerator table to commit the changes.

Figure 8.6 shows the Properties dialog box for this accelerator after again clicking the newly entered line. What happens when the user chooses this new menu item, Tools, Options? A dialog box displays. So, tempting as it may be to start connecting this menu to code, it makes more sense to build the dialog box first.

Building the ShowString Dialog Boxes

Chapter 2, “Dialogs and Controls,” introduces dialog boxes. This section builds on that background. ShowString is going to have two custom dialog boxes: one brought up by Tools, Options and also an About dialog box. An About dialog box has been provided by AppWizard, but it needs to be changed a little; you build the Options dialog box from scratch.

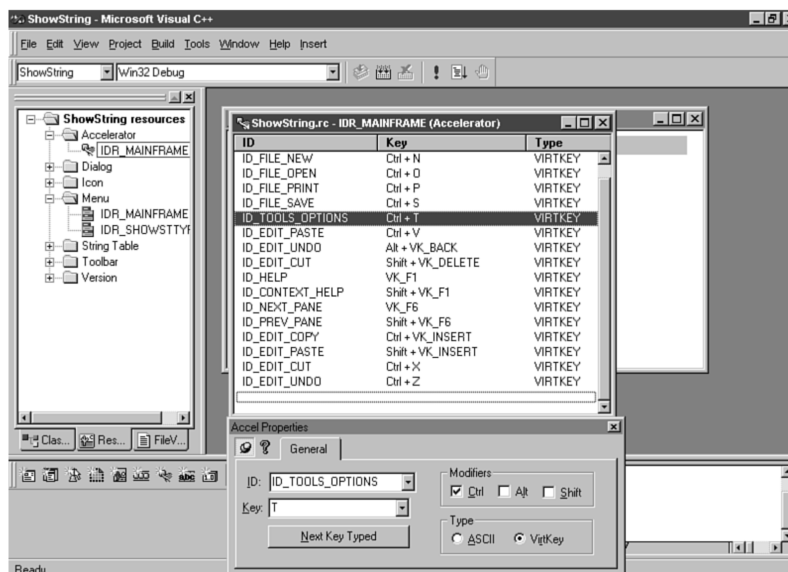


FIG. 8.6 Keyboard accelerators are connected to resource IDs.

ShowString’s About Dialog Box

Figure 8.7 shows the About dialog box that AppWizard makes for you; it contains the application name and the current year. To view the About dialog box for ShowString, click the ResourceView tab in the project workspace window, expand the Dialogs list by clicking the + icon next to the word *Dialogs*, and then double-click IDD_ABOUTBOX to bring up the About dialog box resource.

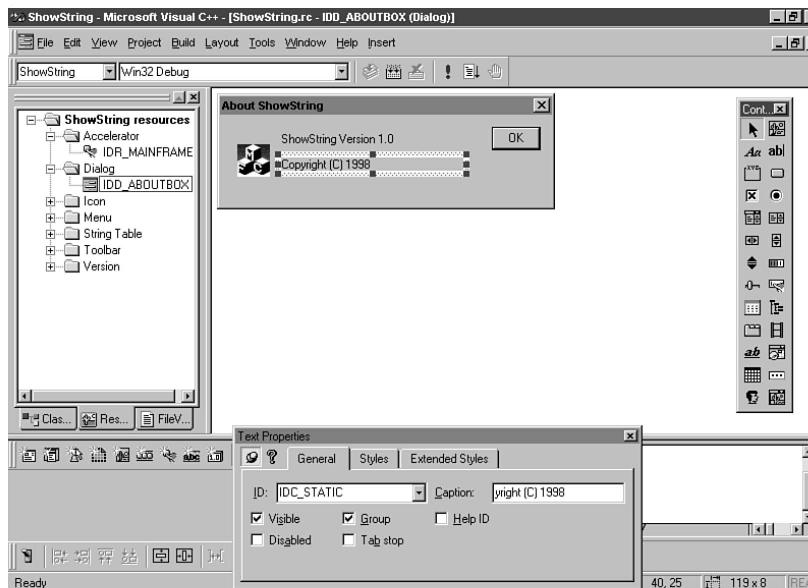


FIG. 8.7 AppWizard makes an About dialog box for you.

You might want to add a company name to your About dialog box. Here's how to add **Que Books**, as an example. Click the line of text that reads Copyright© 1998, and it will be surrounded by a selection box. Bring up the Properties dialog box, if it isn't up. Edit the caption to add **Que Books** at the end; the changes are reflected immediately in the dialog box.

I decided to add a text string to remind users what book this application is from. Here's how to do that:

1. Size the dialog box a little taller by clicking the whole dialog box to select it, clicking the sizing square in the middle of the bottom border, and dragging the bottom border down a little. (This visual editing is what gave Visual C++ its name when it first came out.)

2. In the floating toolbar called Controls, click the button labeled *Aa* to get a *static control,* which means a piece of text that the user cannot change, perfect for labels like this. Click within the dialog box under the other text to insert the static text there.

3. In the Properties dialog box, change the caption from Static to **Using Visual C++ 6**. The box automatically resizes to fit the text.

4. Hold down the Ctrl key and click the other two static text lines in the dialog box. Choose Layout, Align Controls, Left, which aligns the edges of the three selected controls. The one you select last stays still, and the others move to align with it.

5. Choose Layout, Space Evenly, Down. These menu options can save you a great deal of dragging, squinting at the screen, and then dragging again.

The About dialog box will resemble Figure 8.8.

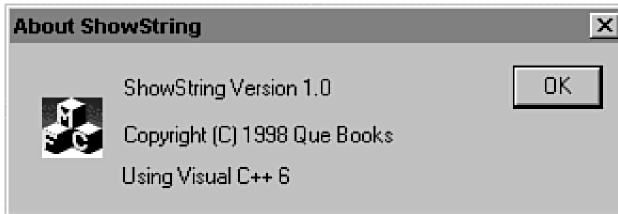


FIG. 8.8 In a matter of minutes, you can customize your About dialog box.

ShowString's Options Dialog Box

The Options dialog box is simple to build. First, make a new dialog box by choosing Insert, Resource and then double-clicking Dialog. An empty dialog box called Dialog1 appears, with an OK button and a Cancel button, as shown in Figure 8.9.

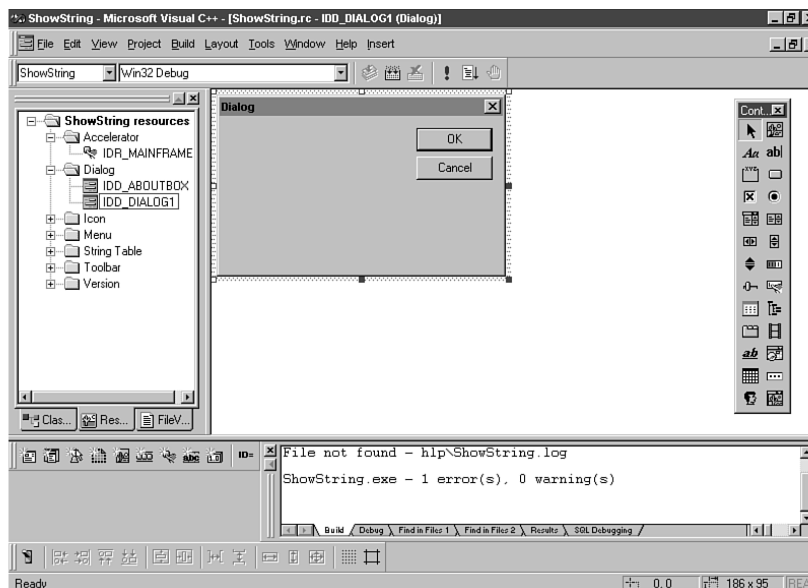


FIG. 8.9 A new dialog box always has OK and Cancel buttons.

Next, follow these steps to convert the empty dialog box into the Options dialog box:

1. Change the ID to IDD_OPTIONS and the caption to Options.

2. In the floating toolbar called Controls, click the button labeled ab| to get an edit box in which the user can enter the new value for the string. Click inside the dialog box to place the control and then change the ID to IDC_OPTIONS_STRING. (Control IDs should all start with IDC and then mention the name of their dialog box and an identifier that is unique to that dialog box.)

3. Drag the sizing squares to resize the edit box as wide as possible.

4. Add a static label above the edit box and change that caption to **String:**.

You will revisit this dialog box later, when adding the appearance capabilities, but for now it's ready to be connected. It will look like Figure 8.10.

FIG. 8.10 The Options dialog box is the place to change the string.

Making the Menu Work

When the user chooses Tools, Options, the Options dialog box should display. You use ClassWizard to arrange for one of your functions to be called when the item is chosen, and then you write the function, which creates an object of your dialog box class and then displays it.

The Dialog Box Class

ClassWizard makes the dialog box class for you. While the window displaying the IDD_OPTIONS dialog box has focus, choose View, ClassWizard. ClassWizard realizes there is not yet a class that corresponds to this dialog box and offers to create one, as shown in Figure 8.11.

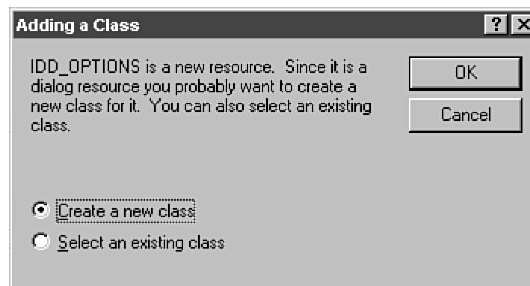


FIG. 8.11 Create a C++ class to go with the new dialog box.

Leave Create a New Class selected and then click OK. The New Class dialog box, shown in Figure 8.12, appears.

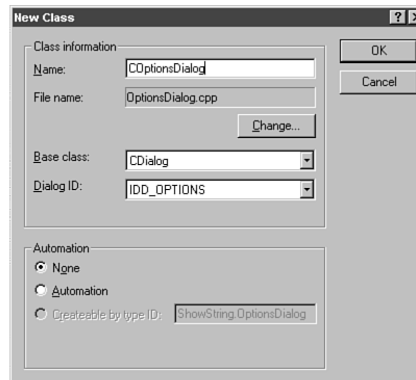


FIG. 8.12 The dialog box class inherits from CDialog.

Fill in the dialog box as follows:

1. Choose a sensible name for the class, one that starts with C and contains the word Dialog; this example uses COptionsDialog.

2. The base class defaults to CDialog, which is perfect for this case.

3. Click OK to create the class.

The ClassWizard dialog box has been waiting behind these other dialog boxes, and now you use it. Click the Member Variables tab and connect IDC_OPTIONS_STRING to a CString called m_string, just as you connected controls to member variables of the dialog box class in Chapter 2. Click OK to close ClassWizard.

Perhaps you're curious about what code was created for you when ClassWizard made theclass. The header file is shown in Listing 8.5.

Listing 8.5 OPTIONSDIALOG.H—Header File for COptionsDialog

```
// OptionsDialog.h : header file

//

/////////////////////////////////////////////////////////////////

// COptionsDialog dialog

class COptionsDialog : public CDialog

{

// Construction
```

```
public:
CODtionsDialog(CWnd* pParent = NULL); // standard constructor
// Dialog Data
    //{{AFX_DATA(COptionsDialog)
    enum { IDD = IDD_OPTIONS };
    CString m_string;
    //}}AFX_DATA
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(COptionsDialog)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV
    support
    //}}AFX_VIRTUAL
// Implementation
protected:
    // Generated message map functions
    //{{AFX_MSG(COptionsDialog)
        // NOTE: The ClassWizard will add member functions here
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

There are an awful lot of comments here to help ClassWizard find its way around in the file when the time comes to add more functionality, but there is only one member variable, m_string; one constructor; and one member function, DoDataExchange(), which gets the control value into the member variable, or vice versa. The source file isn't much longer; it's shown in Listing 8.6.

Listing 8.6 OPTIONSDIALOG.CPP—Implementation File for COptionsDialog

```
// OptionsDialog.cpp : implementation file

//

#include "stdafx.h"

#include "ShowString.h"

#include "OptionsDialog.h"

#ifdef _DEBUG

#define new DEBUG_NEW

#undef THIS_FILE

static char THIS_FILE[] = __FILE__;

#endif

///////////////////////////////////////////////////////////////////////////
/

// COptionsDialog dialog

COptionsDialog::COptionsDialog(CWnd* pParent /*=NULL*/)

        : CDialog(COptionsDialog::IDD, pParent)

{

        //{{AFX_DATA_INIT(COptionsDialog)

        m_string = _T("");

        //}}AFX_DATA_INIT

}

void COptionsDialog::DoDataExchange(CDataExchange* pDX)

{

        CDialog::DoDataExchange(pDX);

        //{{AFX_DATA_MAP(COptionsDialog)

        DDX_Text(pDX, IDC_OPTIONS_STRING, m_string);
```

```
	//}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(COptionsDialog, CDialog)
	//{{AFX_MSG_MAP(COptionsDialog)
	// NOTE: The ClassWizard will add message map macros here
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

The constructor sets the string to an empty string; this code is surrounded by special ClassWizard comments that enable it to add other variables later. The DoDataExchange() function calls DDX_Text() to transfer data from the control with the resource ID IDC_OPTIONS_STRING to the member variable m_string, or vice versa. This code, too, is surrounded by ClassWizard comments. Finally, there is an empty message map because COptionsDialog doesn't catch any messages.

Catching the Message

The next step in building ShowString is to catch the command message sent when the user chooses Tools, Options. There are seven classes in ShowString: CAboutDlg, CChildFrame, CMainFrame, COptionsDialog, CShowStringApp, CShowStringDoc, and CShowStringView. Which one should catch the command? The string and the options will be saved in the document and displayed in the view, so one of those two classes should handle the changing of the string. The document owns the private variable and will not let the view change the string unless you implement a public function to set the string. So, it makes the most sense to have the document catch the message.

To catch the message, follow these steps:

1. Open ClassWizard (if it isn't already open).

2. Click the Message Maps tab.

3. Select CShowStringDoc from the Class Name drop-down list box.

4. Select ID_TOOLS_OPTIONS from the Object IDs list box on the left, and select COMMAND from the Messages list box on the right.

5. Click Add Function to add a function to handle this command.

6. The Add Member Function dialog box, shown in Figure 8.13, appears, giving you an opportunity to change the function name from the usual one. Do not change it; just click OK.

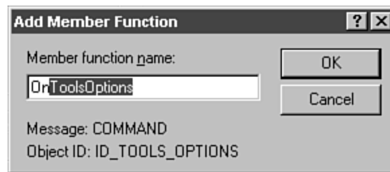


FIG. 8.13 ClassWizard suggests a good name for the message-catching function.

Click Edit Code to close ClassWizard and edit the newly added function. What happened to CShowStringDoc when you arranged for the ID_TOOLS_OPTIONS message to be caught? The new message map in the header file is shown in Listing 8.7.

Listing 8.7 SHOWSTRINGDOC.H—Message Map for CShowStringDoc

```
// Generated message map functions

protected:

    //{{AFX_MSG(CShowStringDoc)

    afx_msg void OnToolsOptions();

    //}}AFX_MSG

DECLARE_MESSAGE_MAP()
```

This is just declaring the function. In the source file, ClassWizard changed the message maps shown in Listing 8.8.

Listing 8.8 SHOWSTRINGDOC.CPP—Message Map for CShowStringDoc

```
BEGIN_MESSAGE_MAP(CShowStringDoc, CDocument)
//{{AFX_MSG_MAP(CShowStringDoc)
ON_COMMAND(ID_TOOLS_OPTIONS, OnToolsOptions)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

This arranges for OnToolsOptions() to be called when the command ID_TOOLS_OPTIONS is sent. ClassWizard also added a skeleton for OnToolsOptions():

```
void CShowStringDoc::OnToolsOptions()
{
    // TODO: Add your command handler code here
}
```

Making the Dialog Box Work

OnToolsOptions() should initialize and display the dialog box and then do something with the value that the user provided. (This process was first discussed in Chapter 2. You have already connected the edit box to a member variable, m_string, of the dialog box class. You initialize this member variable before displaying the dialog box and use it afterwards. OnToolsOptions(), shown in Listing 8.9, displays the dialog box. Add this code to the empty function ClassWizard generated for you when you arranged to catch the message.

Listing 8.9 SHOWSTRINGDOC.CPP—OnToolsOptions()

```
void CShowStringDoc::OnToolsOptions()
{
    COptionsDialog dlg;
    dlg.m_string = string;
    if (dlg.DoModal() == IDOK)
    {
        string = dlg.m_string;
        SetModifiedFlag();
        UpdateAllViews(NULL);
    }
}
```

This code fills the member variable of the dialog box with the document's member variable (ClassWizard added m_string as a public member variable of COptionsDialog, so the document can change it) and then brings up the dialog box by calling DoModal(). If the user clicks OK, the member variable of the document changes, the modified flag is set (so that the user is prompted to save the document on exit), and the view is asked to redraw itself with a call to UpdateAllViews(). For this to compile, of course, the compiler must know what a COptionsDialog is, so add this line at the beginning of ShowStringDoc.cpp:

```
#include "OptionsDialog.h"
```

At this point, you can build the application and run it. Choose Tools, Options and change the string. Click OK and you see the new string in the view. Exit the application; you are asked whether to save the file. Save it, restart the application, and open the file again. The default "Hello, world!" document remains open, and the changed document is open with a different string. The application works, as you can see in Figure 8.14 (the windows are resized to let them both fit in the figure).

FIG. 8.14 ShowString can change the string, save it to a file, and reload it.

Adding Appearance Options to the Options Dialog Box

ShowString doesn't have much to do, just demonstrate menus and dialog boxes. However, the only dialog box control that ShowString uses is an edit box. In this section, you add a set of radio buttons and check boxes to change the way the string is drawn in the view.

Changing the Options Dialog Box

It is quite simple to incorporate a full-fledged Font dialog box into an application, but the example in this section is going to do something much simpler. A group of radio buttons will give the user a choice of several colors. One check box will enable the user to specify that the text should be centered horizontally, and another that the text be centered vertically. Because these are check boxes, the text can be either, neither, or both.

Open the IDD_OPTIONS dialog box by double-clicking it in the ResourceView window, and then add the radio buttons by following these steps:

1. Stretch the dialog box taller to make room for the new controls.

2. Click the radio button in the Controls floating toolbar, and then click the Options dialogbox to drop the control.

3. Choose View, Properties and then pin the Properties dialog box in place.

4. Change the resource ID of the first radio button to IDC_OPTIONS_BLACK, and change thecaption to **&Black**.

5. Select the Group box to indicate that this is the first of a group of radio buttons.

6. Add another radio button with resource ID IDC_OPTIONS_RED and **&Red** as the caption. Do not select the Group box because the Red radio button doesn't start a new group but is part of the group that started with the Black radio button.

7. Add a third radio button with resource ID IDC_OPTIONS_GREEN and **&Green** as the caption. Again, do not select Group.

8. Drag the three radio buttons into a horizontal arrangement, and select all three by clicking on one and then holding Ctrl while clicking the other two.

9. Choose Layout, Align Controls, Bottom (to even them up).

10. Choose Layout, Space Evenly, Across to space the controls across the dialog box.

Next, add the check boxes by following these steps:

1. Click the check box in the Controls floating toolbar and then click the Options dialog box, dropping a check box onto it.

2. Change the resource ID of this check box to IDC_OPTIONS_HORIZCENTER and the caption to **Center &Horizontally**.

3. Select the Group box to indicate the start of a new group after the radio buttons.

4. Drop another check box onto the dialog box as in step 1 and give it the resource ID IDC_OPTIONS_VERTCENTER and the caption **Center &Vertically**.

5. Arrange the check boxes under the radio buttons.

6. Click the Group box on the Controls floating toolbar, and then click and drag a group box around the radio buttons. Change the caption to **Text Color**.

7. Move the OK and Cancel buttons down to the bottom of the dialog box.

8. Select each horizontal group of controls and use Layout, Center in Dialog, Horizontal to make things neater.

9. Choose Edit, Select All, and then drag all the controls up toward the top of the dialog box. Shrink the dialog box to fit around the new controls. It should now resemble Figure 8.15.

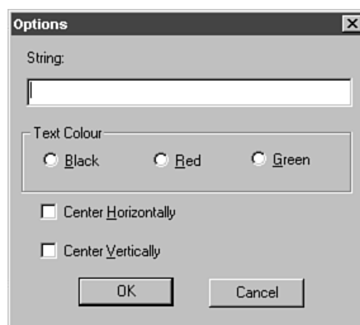


FIG. 8.15 The Options dialog box for ShowString has been expanded.

Finally, set the tab order by choosing Layout, Tab Order and then clicking the controls, in this order:

1. IDC_OPTIONS_STRING
2. IDC_OPTIONS_BLACK
3. IDC_OPTIONS_RED
4. IDC_OPTIONS_GREEN
5. IDC_OPTIONS_HORIZCENTER
6. IDC_OPTIONS_VERTCENTER
7. IDOK
8. IDCANCEL

Then click away from the dialog box to leave the two static text controls as positions 9 and 10.

Adding Member Variables to the Dialog Box Class

Having added controls to the dialog box, you need to add corresponding member variables to the COptionsDialog class. Bring up ClassWizard, select the Member Variable tab, and add member variables for each control. Figure 8.16 shows the summary of the member variables created. The check boxes are connected to BOOL variables; these member variables are TRUE if the box is selected and FALSE if it isn't. The radio buttons are handled differently. Only the first—the one with the Group box selected in its Properties dialog box—is connected to a member variable. That integer is a zero-based index that indicates which button is selected. In other words, when the Black button is selected, m_color is 0; when Red is selected, m color is 1; and when Green is selected, m_color is 2.

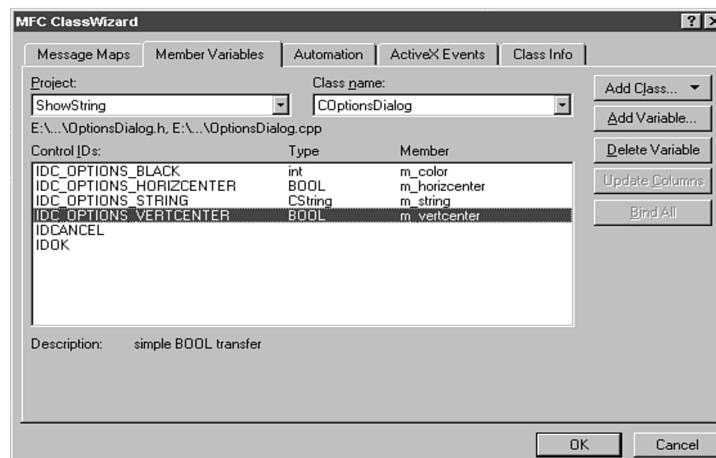


FIG. 8.16 Member variables in the dialog box class are connected to individual controls or the group of radio buttons.

Adding Member Variables to the Document

The variables to be added to the document are the same ones that were added to the dialog box. You add them to the CShowStringDoc class definition in the header file, to OnNewDocument(), and to Serialize(). Add the lines in Listing 8.10 at the top of the CShowStringDoc definition in ShowStringDoc.h, replacing the previous definition of string and GetString(). Make sure that the variables are private and the functions are public.

Listing 8.10 SHOWSTRINGDOC.H—CShowStringDoc Member Variables

```
private:
    CString string;
    int color;
    BOOL horizcenter;
    BOOL vertcenter;
public:
    CString GetString() {return string;}
    int GetColor() {return color;}
    BOOL GetHorizcenter() {return horizcenter;}
    BOOL GetVertcenter() {return vertcenter;}
```

As with string, these are private variables with public get functions but no set functions. All these options should be serialized; the new Serialize() is shown in Listing 8.11. Change your copy by double-clicking the function name in ClassView and adding the new code.

Listing 8.11 SHOWSTRINGDOC.CPP—Serialize()

```
void CShowStringDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << string;
        ar << color;
```

```
        ar << horizcenter;
        ar << vertcenter;
    }
    else
    {
        ar >> string;
        ar >> color;
        ar >> horizcenter;
        ar >> vertcenter;
    }
}
```

Finally, you need to initialize these variables in OnNewDocument(). What are good defaults for these new member variables? Black text, centered in both directions, was the old behavior, and it makes sense to use it as the default. The new OnNewDocument() is shown in Listing 8.12.

Listing 8.12 SHOWSTRINGDOC.CPP—OnNewDocument()

```
BOOL CShowStringDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
    return FALSE;
    string = "Hello, world!";
    color = 0; //black
    horizcenter = TRUE;
    vertcenter = TRUE;
    return TRUE;
}
```

Of course, at the moment, users cannot change these member variables from the defaults. To allow the user to change the variables, you have to change the function that handles the dialog box.

Changing *OnToolsOptions()*

The OnToolsOptions() function sets the values of the dialog box member variables from the document member variables and then displays the dialog box. If the user clicks OK, the document member variables are set from the dialog box member variables and the view is redrawn. Having just added three member variables to the dialog box and the document, you have three lines to add before the dialog box displays and then three more to add in the block that's called after OK is clicked. The new OnToolsOptions() is shown in Listing 8.13.

Listing 8.13 SHOWSTRINGDOC.CPP—OnToolsOptions()

```
void CShowStringDoc::OnToolsOptions()
{
    COptionsDialog dlg;
    dlg.m_string = string;
    dlg.m_color = color;
    dlg.m_horizcenter = horizcenter;
    dlg.m_vertcenter = vertcenter;
    if (dlg.DoModal() == IDOK)
    {
        string = dlg.m_string;
        color = dlg.m_color;
        horizcenter = dlg.m_horizcenter;
        vertcenter = dlg.m_vertcenter;
        SetModifiedFlag();
        UpdateAllViews(NULL);
    }
}
```

What happens when the user opens the dialog box and changes the value of a control, say, by deselecting Center Horizontally? The framework—through Dialog Data Exchange (DDX), as set up by ClassWizard—changes the value of COptionsDialog::m_horizcenter to FALSE. This code in OnToolsOptions() changes the value of CShowStringDoc::horizcenter to FALSE. When the user saves the document, Serialize() saves horizcenter. This is all good, but none of this code actually changes the way the view is drawn. That involves OnDraw().

Changing *OnDraw()*

The single call to DrawText() in OnDraw() becomes a little more complex now. The document member variables are used to set the view's appearance. Edit OnDraw() by expanding CShowStringView in the ClassView and double-clicking OnDraw().

The color is set with CDC::SetTextColor() before the call to DrawText(). You should always save the old text color and restore it when you are finished. The parameter to SetTextColor() is a COLORREF, and you can directly specify combinations of red, green, and blue as hex numbers in the form 0x00bbggrr, so that, for example, 0x000000FF is bright red. Most people prefer to use the RGB macro, which takes hex numbers from 0x0 to 0xFF, specifying the amount of each color; bright red is RGB(FF,0,0), for instance. Add the lines shown in Listing 8.14 before the call to DrawText() to set up everything.

Listing 8.14 SHOWSTRINGDOC.CPP—OnDraw() Additions Before DrawText() Call

```
COLORREF oldcolor;

    switch (pDoc->GetColor())

    {

        case 0:

                oldcolor = pDC->SetTextColor(RGB(0,0,0)); //black

        break;

        case 1:

                oldcolor = pDC->SetTextColor(RGB(0xFF,0,0)); //red

        break;

        case 2:

                oldcolor = pDC->SetTextColor(RGB(0,0xFF,0)); //green

        break;

    }
```

Add this line after the call to DrawText():

```
pDC->SetTextColor(oldcolor);
```

There are two approaches to setting the centering flags. The brute-force way is to list the four possibilities (neither, horizontal, vertical, and both) and have a different DrawText() statement for each. If you were to add other settings, this would quickly become unworkable. It’s better to set up an integer to hold the DrawText() flags and OR in each flag, if appropriate. Add the lines shown in Listing 8.15 before the call to DrawText().

Listing 8.15 SHOWSTRINGDOC.CPP—OnDraw() Additions After DrawText() Call

```
int DTflags = 0;

if (pDoc->GetHorizcenter())

{

    DTflags |= DT_CENTER;

}

if (pDoc->GetVertcenter())

{

    DTflags |= (DT_VCENTER|DT_SINGLELINE);

}
```

The call to DrawText() now uses the DTflags variable:

```
pDC->DrawText(pDoc->GetString(), &rect, DTflags);
```

Now the settings from the dialog box have made their way to the dialog box class, to the document, and finally to the view, to actually affect the appearance of the text string. Build and execute ShowString and then try it. Any surprises? Be sure to change the text, experiment with various combinations of the centering options, and try all three colors.

UNIT - IV

STATUS BARS AND TOOLBARS

In this chapter

Working with Toolbars

Working with Status Bars

Working with Rebars

Building a good user interface is half the battle of programming a Windows application. Luckily, Visual C++ and its AppWizard supply an amazing amount of help in creating an application that supports all the expected user-interface elements, including menus, dialog boxes, toolbars, and status bars. The subjects of menus and dialog boxes are covered in Chapters 2, "Dialogs and Controls," and 8, "Building a Complete Application: ShowString." In this chapter, you learn how to get the most out of toolbars and status bars.

Working with Toolbars

The buttons on a toolbar correspond to commands, just as the items on a menu do. Although you can add a toolbar to your application with AppWizard, you still need to use a little programming polish to make things just right. This is because every application is different and AppWizard can create only the most generally useful toolbar for most applications. When you create your own toolbars, you will probably want to add or delete buttons to support your application's unique command set.

For example, when you create a standard AppWizard application with a toolbar, AppWizard creates the toolbar shown in Figure 9.1. This toolbar provides buttons for the commonly used commands in the File and Edit menus, as well as a button for displaying the About dialog box. What if your application doesn't support these commands? It's up to you to modify the default toolbar to fit your application.

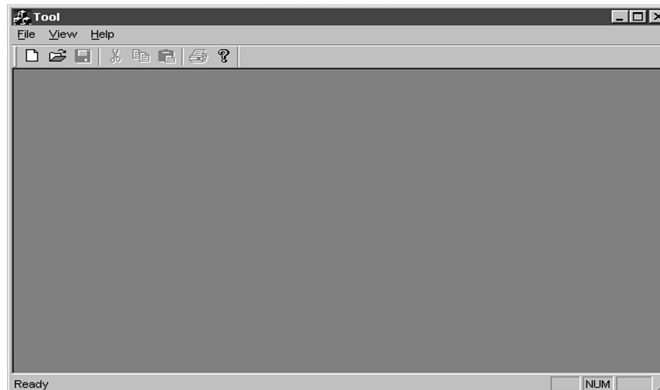


FIG. 9.1 The default toolbar provides buttons for commonly used commands.

Deleting Toolbar Buttons

Create a multiple document interface application with a toolbar by choosing File, New; selecting the Project tab; highlighting MFC AppWizard (exe); naming the application **Tool**; and accepting the defaults in every dialog box. If you like, you can click the Finish button in step 1 to speed up the process. AppWizard provides a docking toolbar by default. Build and run theapplication, and you should see a toolbar of your own, just like Figure 9.1.

Before moving on, play with this toolbar a little. On the View menu, you can toggle whether the toolbar is displayed. Turn it off and then on again. Now click and hold on the toolbar between buttons and pull it down into the working area of your application. Let it go, and it's a floating palette. Drag it around and drop it at the bottom of the application or one of the sides—it will dock against any side of the main window. Watch the tracking rectangle change shape to show you it will dock if you drop it. Drag it back off again so that it's floating and close it by clicking the small x in the upper-right corner. Bring it back with the View menu and notice that it comes back right where you left it. All this functionality is yours free from AppWizard and MFC.

The first step in modifying the toolbar is to delete buttons you no longer need. To do this, first select the ResourceView tab to display your application's resources by clicking on the + next to Tool Resources. Click the + next to Toolbar and double-click the IDR_MAINFRAME toolbar resource to edit it, as shown in Figure 9.2. (The Graphics and Colors palettes, shown floating in Figure 9.2, are docked by default. You can move them around by grabbing the wrinkles at the top.)

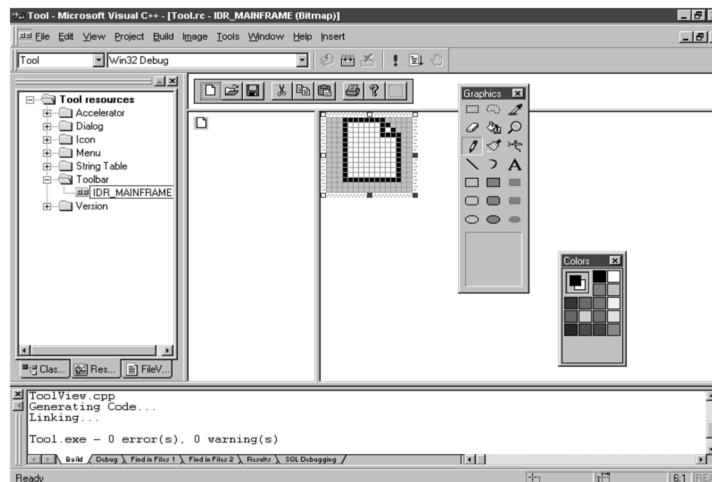


FIG. 9.2 Use the toolbar editor to customize your application's toolbar

After you have the toolbar editor on the screen, deleting buttons is as easy as dragging the unwanted buttons from the toolbar. Place your mouse pointer on the button, hold down the left mouse button, and drag the unwanted button away from the toolbar. When you release the mouse button, the toolbar button disappears. In the Tool application, delete all the buttons except the Help button with a yellow question mark. Figure 9.3 shows the edited toolbar with only the Help button remaining. The

single blank button template is only a starting point for the next button you want to create. If you leave it blank, it doesn't appear in the final toolbar.

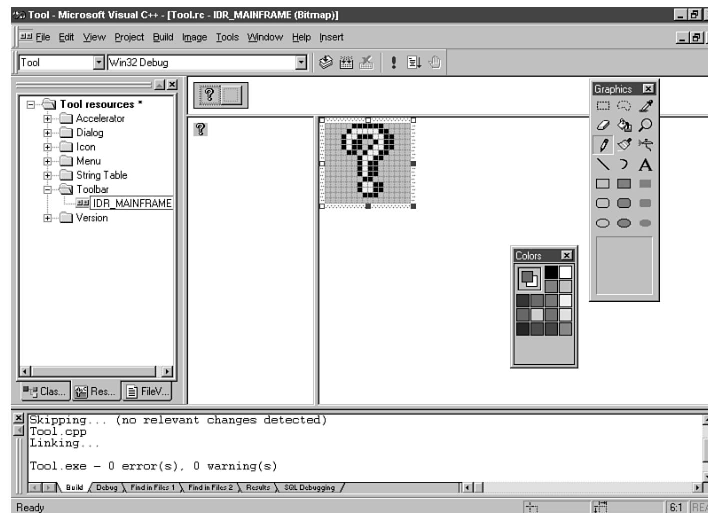


FIG. 9.3 This edited toolbar has only a single button left (not counting the blank button template).

Adding Buttons to a Toolbar

Adding buttons to a toolbar is a two-step process: First you draw the button's icon, and then you match the button with its command. To draw a new button, first click the blank button template in the toolbar. The blank button appears enlarged in the edit window, as shown in Figure 9.4.

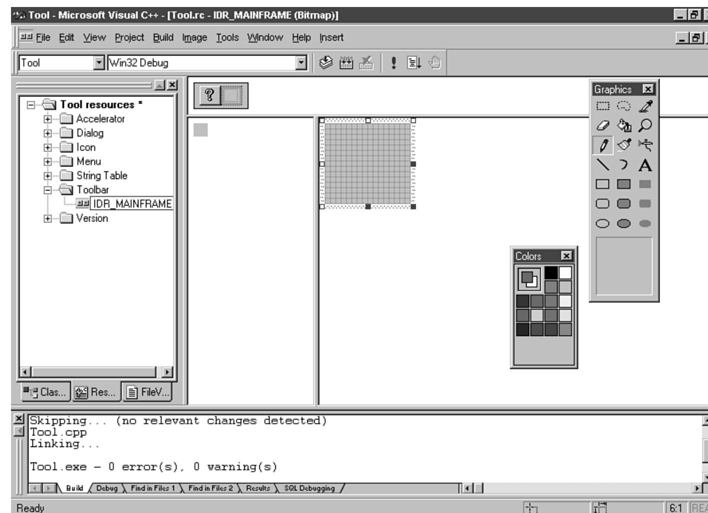


FIG. 9.4 Click the button template to open it in the button editor.

Suppose you want to create a toolbar button that draws a red circle in the application's window. Draw a red circle on the blank button with the Ellipse tool, and you've created the button's icon. Open the properties box and give the button an appropriate ID, such as ID_CIRCLE in this case.

Now you need to define the button's description and ToolTip. The description appears in the application's status bar. In this case, a description of "Draws a red circle in the window" might be good. The ToolTip appears whenever the user leaves the mouse pointer over the button for a second or two, acting as a reminder of the button's purpose. A ToolTip of *Circle* would be appropriate for the circle button. Type these two text strings into the Prompt box. The description comes first, followed by the newline character (\n) and the ToolTip, as shown in Figure 9.5.

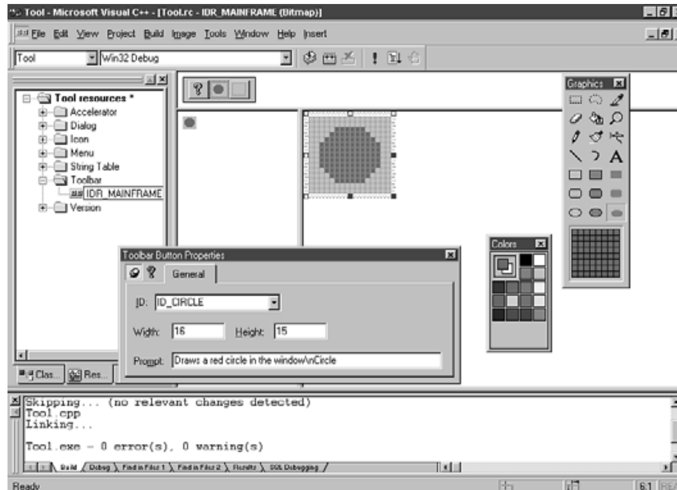


FIG. 9.5 After drawing the button, specify its properties.

You've now defined a command ID for your new toolbar button. Usually, you use the command ID of an existing menu item already connected to some code. In these cases, simply choose the existing command ID from the drop-down box, and your work is done. The prompt is taken from the properties of the menu item, and the message handler has already been arranged for the menu item. You will already be handling the menu item, and that code will handle the toolbar click, too. In this application, the toolbar button doesn't mirror a menu item, so you will associate the ID with a message-handler function that MFC automatically calls when the user clicks the button.

To do this, follow these steps:

1. Make sure the button for which you want to create a message handler is selected in the custom toolbar, and then open ClassWizard. Working with Toolbars

2. The MFC ClassWizard property sheet appears, with the button's ID already selected (see Figure 9.6). To add the message-response function, select in the Class Name box the class to which you want to add the function (the sample application uses the view class).

3. Double-click the COMMAND selection in the Messages box.

4. Accept the function name that MFC suggests in the next message box, and you're all set.

Click OK to finalize your changes.

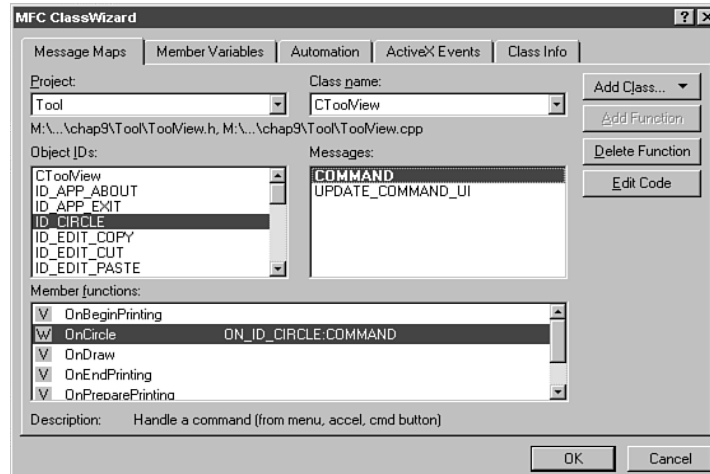


FIG. 9.6 You can use ClassWizard to catch messages from your toolbar buttons

If you compile and run the application now, you will see the window shown in Figure 9.7. In the figure, you can see the new toolbar button, as well as its ToolTip and description line. The toolbar looks sparse in this example, but you can add as many buttons as you like. You can create as many buttons as you need; just follow the same procedure for each. After you have created the buttons, you're through with the toolbar resources and ready to write the code that responds to the buttons. For example, in the previous example, a circle button was added to the toolbar, and a message-response function, called OnCircle(), was added to the program. MFC calls that message-response function whenever the user clicks the associated button. However, right now, that function doesn't do anything, as shown in Listing 9.1.

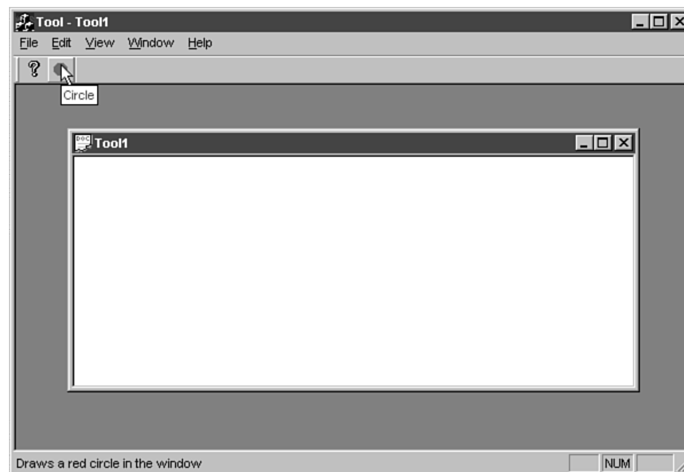


FIG. 9.7 The new toolbar button shows its ToolTip and description.

Listing 9.1 An Empty Message-Response Function

```
void CToolView::OnCircle()
```

```
{
        // TODO: Add your command handler code here
}
```

Although the circle button is supposed to draw a red circle in the window, you can see that the OnCircle() function is going to need a little help accomplishing that task. Add the lines shown in Listing 9.2 to the function so that the circle button will do what it's supposed to do, as shown in Figure 9.8. This drawing code makes a brush, selects it into the DC, draws an ellipse with it, and then restores the old brush. The details of drawing are discussed in Chapter 5, "Drawing on the Screen."

Listing 9.2 CToolView::OnCircle()

```
void CToolView::OnCircle()
{
        CClientDC clientDC(this);
        CBrush newBrush(RGB(255,0,0));
        CBrush* oldBrush = clientDC.SelectObject(&newBrush);
        clientDC.Ellipse(20, 20, 200, 200);
        clientDC.SelectObject(oldBrush);
}
```

The *CToolBar* Class's Member Functions

In most cases, after you have created your toolbar resource and associated its buttons with the appropriate command IDs, you don't need to bother any more with the toolbar. The code generated by AppWizard creates the toolbar for you, and MFC takes care of calling the buttons' response functions for you. However, at times you might want to change the toolbar's default behavior or appearance in some way. In those cases, you can call on the CToolBar class's member functions, which are listed in Table 9.1 along with their descriptions. The toolbar is accessible from the CMainFrame class as the m_wndToolBar member variable. Usually, you change the toolbar behavior in CMainFrame::OnCreate().

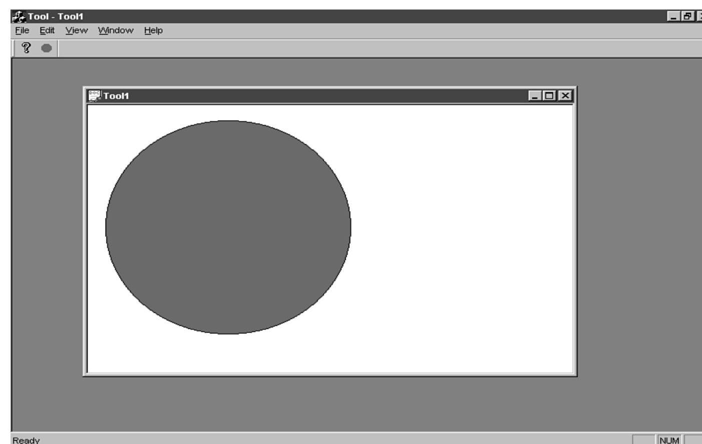


FIG. 9.8 After adding code to OnCircle(), the new toolbar button actually does something.

Table 9.1 Member Functions of the CToolBar Class

Function	Description
CommandToIndex()	Obtains the index of a button, given its ID
Create()	Creates the toolbar
GetButtonInfo()	Obtains information about a button
GetButtonStyle()	Obtains a button's style
GetButtonText()	Obtains a button's text label
GetItemID()	Obtains the ID of a button, given its index
GetItemRect()	Obtains an item's display rectangle, given its index
GetToolBarCtrl()	Obtains a reference to the CToolBarCtrl object represented by the CToolBar object
LoadBitmap()	Loads the toolbar's button images
LoadToolBar()	Loads a toolbar resource
SetBitmap()	Sets a new toolbar button bitmap
SetButtonInfo()	Sets a button's ID, style, and image number
SetButtons()	Sets the IDs for the toolbar buttons
SetButtonStyle()	Sets a button's style
SetButtonText()	Sets a button's text label
SetHeight()	Sets the toolbar's height
SetSizes()	Sets the button sizes

Normally, you don't need to call the toolbar's methods, but you can achieve some unusual results when you do, such as the extra high toolbar shown in Figure 9.9. (The buttons are the same size, but the toolbar window is bigger.) This toolbar resulted from a call to the toolbar object's SetHeight() member function. The CToolBar class's member functions enable you to perform this sort of toolbar trickery, but use them with great caution.

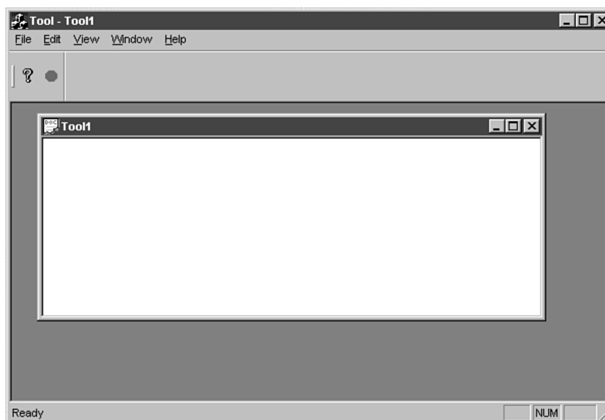


FIG. 9.9 You can use a toolbar object's member functions to change how the toolbar looks and acts.

Working with Status Bars

Status bars are mostly benign objects that sit at the bottom of your application's window, doing whatever MFC instructs them to do. This consists of displaying command descriptions and the status of various keys on the keyboard, including the Caps Lock and Scroll Lock keys. In fact, status bars are so mundane from the programmer's point of view (at least they are in an AppWizard application) that they aren't even represented by a resource that you can edit like a toolbar. When you tell AppWizard to incorporate a status bar into your application, there's not much left for you to do.

Or is there? A status bar, just like a toolbar, must reflect the interface needs of your specific application. For that reason, the CStatusBar class features a set of methods with which you can customize the status bar's appearance and operation. Table 9.2 lists the methods along with brief descriptions.

Table 9.2 Methods of the CStatusBar Class

Method	Description
CommandToIndex()	Obtains an indicator's index, given its ID
Create()	Creates the status bar
GetItemID()	Obtains an indicator's ID, given its index
GetItemRect()	Obtains an item's display rectangle, given its index
GetPaneInfo()	Obtains information about an indicator
GetPaneStyle()	Obtains an indicator's style
GetPaneText()	Obtains an indicator's text
GetStatusBarCtrl()	Obtains a reference to the CStatusBarCtrl object represented by the CStatusBar object
SetIndicators()	Sets the indicators' IDs
SetPaneInfo()	Sets the indicators' IDs, widths, and styles
SetPaneStyle()	Sets an indicator's style
SetPaneText()	Sets an indicator's text

When you create a status bar as part of an AppWizard application, you see a window similar to that shown in Figure 9.10. (To make your own, create a project called **Status** and accept all the defaults, as you did for the Tool application.) The status bar has several parts, called *panes*, that display certain information about the status of the application and the system. These panes, which are marked in Figure 9.10, include indicators for the Caps Lock, Num Lock, and Scroll Lock keys, as well as a message area for showing status text and command descriptions. To see a command description, place your mouse pointer over a button on the toolbar (see Figure 9.11).

The most common way to customize a status bar is to add new panes. To add a pane to a status bar, complete these steps:

1. Create a command ID for the new pane.
2. Create a default string for the pane.
3. Add the pane's command ID to the status bar's indicators array.
4. Create a command-update handler for the pane.

The following sections cover these steps in detail.

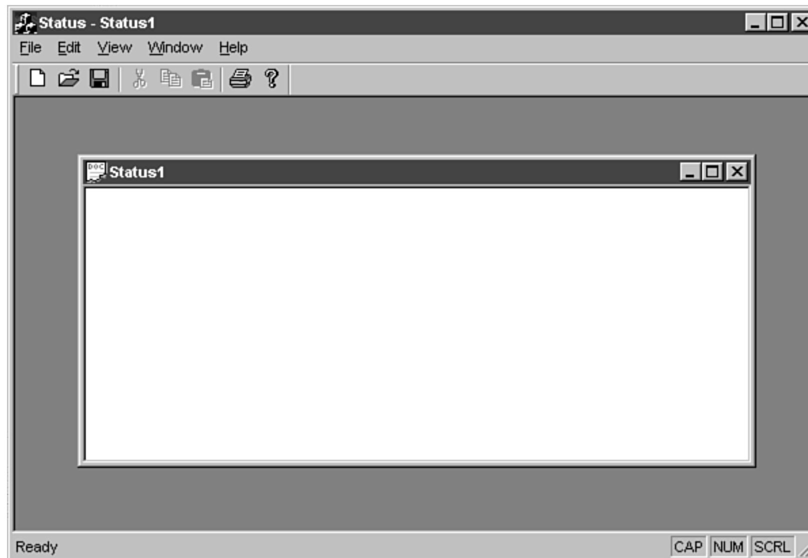


FIG. 9.10 The default MFC status bar contains a number of informative panes.

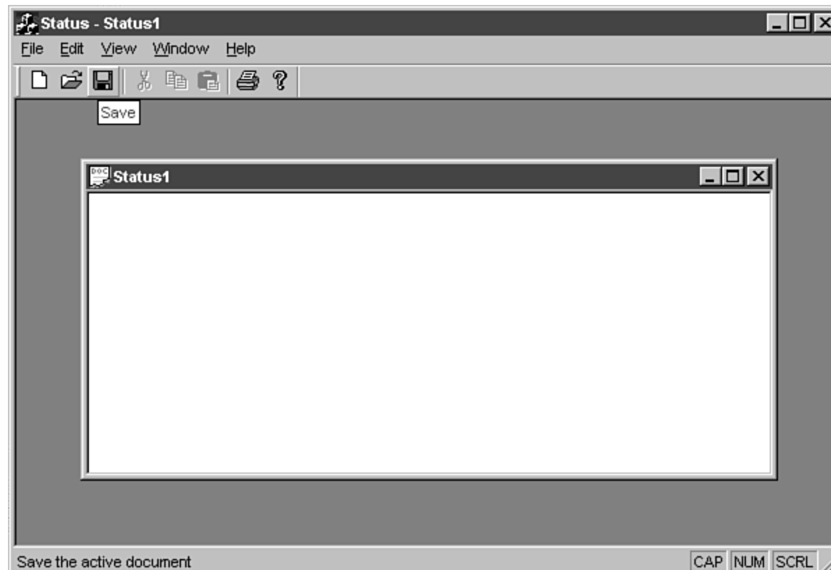


FIG. 9.11 The message area is mainly used for command descriptions

Creating a New Command ID

This step is easy, thanks to Visual C++'s symbol browser. To add the command ID, start by choosing View, Resource Symbols. When you do, you see the Resource Symbols dialog box (see Figure 9.12), which displays the currently defined symbols for your application's resources. Click the New button, and the New Symbol dialog box appears. Type the new ID, ID_MYNEWPANE, into the Name box (see Figure 9.13). Usually, you can accept the value that MFC suggests for the ID.

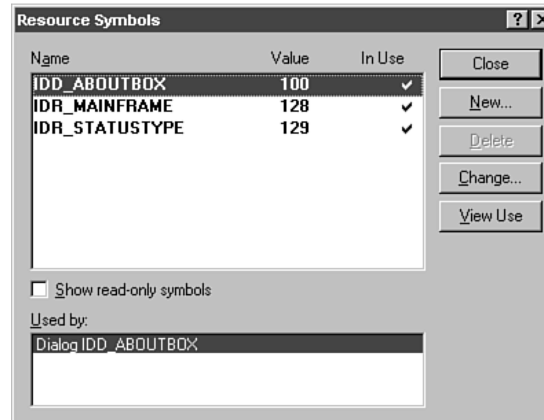


FIG. 9.12 Use the Resource Symbols dialog box to add new command IDs to your application.

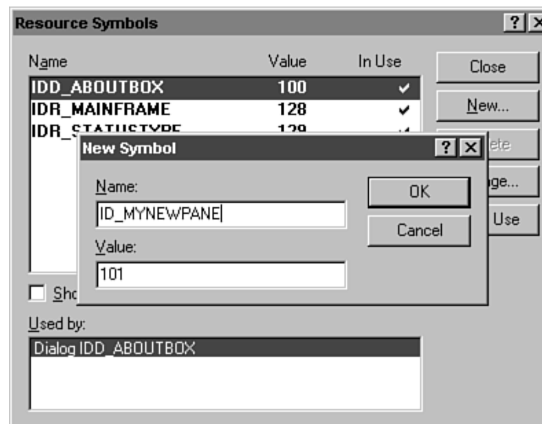


FIG. 9.13 Type the new ID's name and value into the New Symbol dialog box.

Click the OK and Close buttons to finalize your selections, and your new command ID is defined.

Creating the Default String

You have now defined a resource ID, but it isn't being used. To represent a status bar pane, the ID must have a default string defined for it. To define the string, first go to the ResourceView window (by clicking the ResourceView tab in the workspace pane) and double-click the String Table resource to open it in the string

table editor, as shown in Figure 9.14. Now, choose Insert, New String to open the String Properties dialog box. Type the new pane's command ID ID_MYNEWPANE into the ID box (or choose it from the drop-down list) and the default string (**Default string** in this case) into the Caption box (see Figure 9.15).

Adding the ID to the Indicators Array

When MFC constructs your status bar, it uses an array of IDs to determine which panes to display and where to display them. This array of IDs is passed as an argument to the status bar's SetIndicators() member function, which is called in the CMainFrame class's OnCreate() function. You find this array of IDs, shown in Listing 9.3, near the top of the MainFrm.cpp file. One way to reach these lines in the source code editor is to switch to ClassView, expand CMainFrame, double-click OnCreate(), and scroll up one page. Alternatively, you could use FileView to open MainFrm.cpp and scroll down to this code.

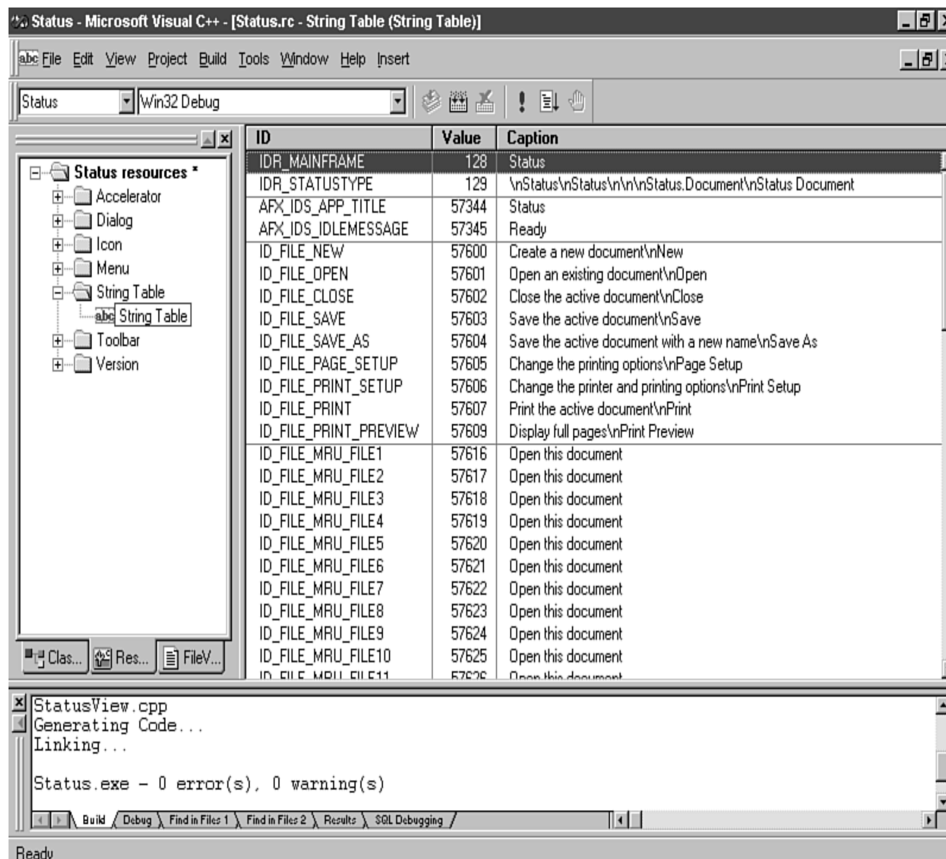


FIG. 9.14 Define the new pane's default string in the string table.

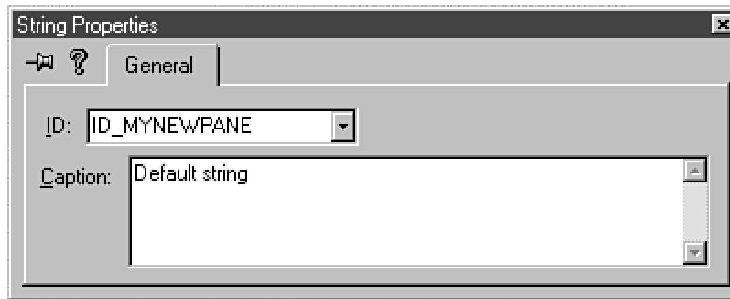


FIG. 9.15 Use the String Properties dialog box to define the new pane's default string.

Listing 9.3 MainFrm.cpp—The Indicator Array

```
static UINT indicators[] =
{
    ID_SEPARATOR, // status line indicator
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
```

To add your new pane to the array, type the pane's ID into the array at the position in which you want it to appear in the status bar, followed by a comma. (The first pane, ID_SEPARATOR, should always remain in the first position.) Listing 9.4 shows the indicator array with the new pane added.

Listing 9.4 MainFrm.cpp—The Expanded Indicator Array

```
static UINT indicators[] =
{
    ID_SEPARATOR, // status line indicator
    ID_MYNEWPANE,
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
```

Creating the Pane's Command-Update Handler

MFC doesn't automatically enable new panes when it creates the status bar. Instead, you must create a command-update handler for the new pane and enable the pane yourself. (You first learned about command-update handlers in Chapter 4, "Messages and Commands.") Also, for most applications, the string displayed in the

pane is calculated on-the-fly—the default string you defined in an earlier step is only a placeholder.

Normally, you use ClassWizard to arrange for messages to be caught, but ClassWizard doesn't help you catch status bar messages. You must add the handler entries to the message map yourself and then add the code for the handler. You add entries to the message map in the header file and the map in the source file, and you add them outside the special AFX_MSG_MAP comments used by ClassWizard.

Double-click CMainFrame in ClassView to open the header file, and scroll to the bottom. Edit the message map so that it resembles Listing 9.5. When you write your own applications, you will use a variety of function names to update status bar panes, but the rest of the declaration will always be the same.

Listing 9.5 MainFrm.h—Message Map

```
// Generated message map functions
protected:
    //{{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        // NOTE - the ClassWizard will add and remove member
        functions here.
        // DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG
    afx_msg void OnUpdateMyNewPane(CCmdUI *pCmdUI);
    DECLARE_MESSAGE_MAP()
```

Next, you add the handler to the source message map to associate the command ID with the handler. Open any CMainFrame function and scroll upwards until you find the message map; then edit it so that it looks like Listing 9.6.

Listing 9.6 MainFrm.cpp—Message Map

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)
        // NOTE - the ClassWizard will add and remove mapping macros
        here.
    // DO NOT EDIT what you see in these blocks of generated code !
    ON_WM_CREATE()
    //}}AFX_MSG_MAP
    ON_UPDATE_COMMAND_UI(ID_MYNEWPANE, OnUpdateMyNewPane)
END_MESSAGE_MAP()
```

You have now arranged for the CMainFrame member function OnUpdateMyNewPane() to be called whenever the status bar pane ID_MYNEWPANE needs to be updated.

Now you're ready to write the new command-update handler. In the handler, you will enable the new pane and set its contents. Listing 9.7 shows the command-update handler for the new pane; add this code to mainfrm.cpp. As you can see, it uses a member variable called m_paneString. Update handlers should be very quick—the job of making sure that m_paneString holds the right string should be tackled in a function that is called less often.

Listing 9.7 CMainFrame::OnUpdateMyNewPane()

```
void CMainFrame::OnUpdateMyNewPane(CCmdUI *pCmdUI)
{
        pCmdUI->Enable();
        pCmdUI->SetText(m_paneString);
}
```

Setting the Status Bar's Appearance

To add the last touch to your status bar demonstration application, you will want a way to set m_paneString. To initialize it, double-click on the CMainFrame constructor to edit it, and add this line:

```
m_paneString = "Default string";
```

The value you entered in the string table is only to assure Visual Studio that the resource ID you created is in use. Right-click CMainFrame in ClassView and choose Add Member Variable to add m_paneString as a private member variable. The type should be CString.

To set up the status bar for the first time, add these lines to CMainFrame::OnCreate(), just before the return statement:

```
CClientDC dc(this);
SIZE size = dc.GetTextExtent(m_paneString);
int index = m_wndStatusBar.CommandToIndex(ID_MYNEWPANE);
m_wndStatusBar.SetPaneInfo(index,ID_MYNEWPANE, SBPS_POPOUT, size.cx);
```

These lines set the text string and the size of the pane. You set the size of the pane with a call to SetPaneInfo(), which needs the index of the pane and the new size. CommandToIndex() obtains the index of the pane, and GetTextExtent() obtains the size. As a nice touch, the call to SetPaneInfo() uses the SBPS_POPOUT style to create a pane that seems to stick out from the status bar, rather than be indented.

The user will change the string by making a menu selection. Open the IDR_STATUSTYPE menu in the resource editor and add a Change String item to the

File menu. (Working with menus is discussed for the first time in Chapter 8.) Let Developer Studio assign it the resource ID.

ID_FILE_CHANGESTRING.

Open ClassWizard and add a handler for this command; it should be caught by CMainFrame because that's where the m_paneString variable is kept. ClassWizard offers to call the handler OnFileChangestring(), and you should accept this name. Click OK twice to close ClassWizard. Insert a new dialog box into the application and call it IDD_PANEDLG. The title should be **Change Pane String**. Add a single edit box, stretched the full width of the dialog box, and leave the ID as IDC_EDIT1. Add a static text item just above the edit box with the caption **New String:**. With the dialog box open in the resource editor, open ClassWizard. Create a new class for the dialog box called CPaneDlg, and associate the edit control, IDC_EDIT1, with a CString member variable of the dialog class called m_paneString.

Switch to ClassView, expand CMainFrame, and double-click
OnFileChangeString() to edit it.

Add the code shown in Listing 9.8.

Listing 9.8 CMainFrame::OnFileChangestring()

```
void CMainFrame::OnFileChangestring()
{
    CPaneDlg dialog(this);
    dialog.m_paneString = m_paneString;
    int result = dialog.DoModal();
    if (result == IDOK)
    {
        m_paneString = dialog.m_paneString;
        CClientDC dc(this);
        SIZE size = dc.GetTextExtent(m_paneString);
        int index = m_wndStatusBar.CommandToIndex(ID_MYNEWPANE);
        m_wndStatusBar.SetPaneInfo(index,
        ID_MYNEWPANE, SBPS_POPOUT, size.cx);
    }
}
```

This code displays the dialog box, and, if the user exits the dialog box by clicking OK, changes the text string and resets the size of the pane. The code is very similar to the lines you added to OnCreate(). Scroll up to the top of MainFrm.cpp and add this line:

```
#include "panedlg.h"
```

This tells the compiler what the CPaneDlg class is. Build and run the Status application, and you should see the window shown in Figure 9.16. As you can see, the status bar contains an extra panel displaying the text Default string. If you choose File, Change String, a dialog box appears into which you can type a new string for the panel. When you exit the dialog box via the OK button, the text appears in the new panel, and the panel resizes itself to accommodate the new string (see Figure 9.17).

FIG. 9.16 The Status Bar Demo application shows how to add and manage a status bar panel.

Working with Rebars

Rebars are toolbars that contain controls other than toolbar buttons. It was possible to add other controls to normal toolbars in the past, but difficult. With rebars, it's simple. Start by using AppWizard to make a project call ReBar. Accept all the defaults on each step, or click Finish on step 1 to speed the process a little. When the project is generated, double-click CMainFrame in ClassView to edit the header file. This frame holds the open documents and is where a classic toolbar goes. The rebar for this sample will go here, too. Add the rebar as a public member variable:

```
CReBar m_rebar;
```

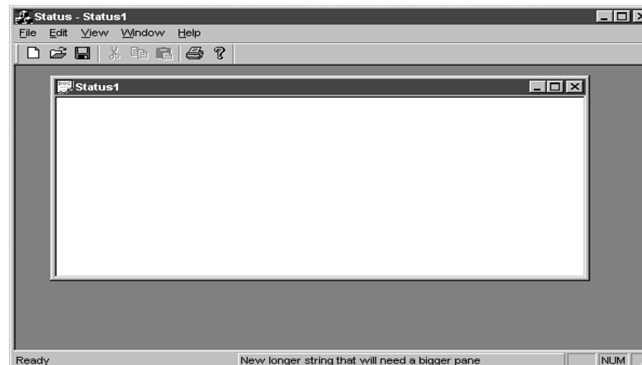


FIG. 9.17 The panel resizes itself to fit the new string.

In this sample application, you will add a check box to the bar—you can add any kind of control at all. A check box, a radio button, and a command button (like the OK or Cancel button on a dialog) are all represented by the CButton class, with slightly different styles. Add the check box to the header file right after the rebar, like this:

```
CButton m_check;
```

You saw in the previous section that an application's toolbar is created and initialized in the OnCreate() function of the mainframe class. The same is true for rebars. Expand CMainFrame in ClassView, and double-click OnCreate() to edit it. Add these lines just before the final return statement:

```
if (!m_rebar.Create(this) )
{
    TRACE0("Failed to create rebar\n");
    return -1; // fail to create
}
```

The check box control will need a resource ID. When you create a control with the dialog editor, the name you give the control is automatically associated with a number. This control will be created in code, so you will have to specify the resource ID yourself, as you did for the new pane in the status bar earlier in this chapter. Choose View, Resource Symbols and click the New button. Type the name IDC_CHECK and accept the number suggested. This adds a line to resource.h, defining IDC_CHECK, and assures you that other controls will not reuse this resource ID.

Back in CMainFrame::OnCreate(), add these lines to create the check box (note the styles

carefully):

```
if (!m_check.Create("Check Here",
        WS_CHILD|WS_VISIBLE|BS_AUTOCHECKBOX,
        CRect(0,0,20,20), this, IDC_CHECK) )
        {
            TRACE0("Failed to create checkbox\n");
            return -1; // fail to create
        }
```

Finally, add this line to add a band containing the check box control to the rebar:

```
m_rebar.AddBar(&m_check, "On The Bar", NULL,
        RBBS_BREAK | RBBS_GRIPPERALWAYS);
```

AddBar() takes four parameters: a pointer to the control that will be added, some text to put next to it, a pointer to a bitmap to use for the background image on the rebar, and a rebar style, made by combining any of these style flags:

- RBBS_BREAK puts the band on a new line, even if there's room for it at the end of an existing line.
- RBBS_CHILDEDGE puts the band against a child window of the frame.
- RBBS_FIXEDBMP prevents moving the bitmap if the band is resized by the user.
- RBBS_FIXEDSIZE prevents the user from resizing the band.
- RBBS_GRIPPERALWAYS guarantees sizing wrinkles are present.
- RBBS_HIDDEN hides the band.
- RBBS_NOGRIPPER suppresses sizing wrinkles.
- RBBS_NOVERT hides the band when the rebar is vertical.
- RBBS_VARIABLEHEIGHT enables the band to be resized by the rebar.

At this point, you can build the project and run it. You should see your rebar, as in Figure 9.18. The check box works in that you can select and deselect it, but nothing happens when you do.

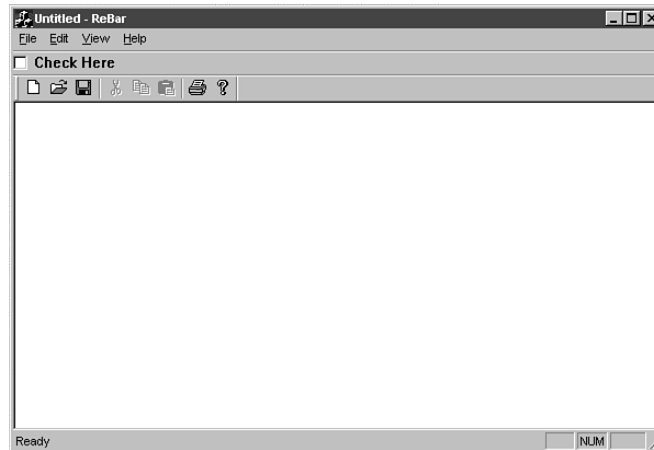


FIG. 9.18 The rebar contains a check box.

To react when the user clicks the button, you need to catch the message and do something based on the message. The simplest thing to do is change what is drawn in the view's OnDraw(), so the view should catch the message. Double click CRebarView in ClassView to edit the header file, and scroll to the message map. Between the closing AFX_MSG and the DECLARE_MESSAGE_MAP, add this line:

```
afx_msg void OnClick();
```

Expand CRebarView in ClassView and double-click OnDraw(), which you will edit in a moment. After it, add this function:

```
void CRebarView::OnClick()
```

```
{
    Invalidate();
}
```

This causes the view to redraw whenever the user selects or deselects the check box. Scroll up in the file until you find the message map, and add (after the three entries related to printing) this line:

```
ON_BN_CLICKED(IDC_CHECK, OnClick)
```

At the top of the file, after the other include statements, add this one:

```
#include "mainFrm.h"
```

Now add these lines to OnDraw() in place of the TODO comment:

```
CString message;
    if ( ((CMainFrame*)(AfxGetApp()->m_pMainWnd))->m_check.GetCheck())
        message = "The box is checked";
    else
        message = "The box is not checked";
        pDC->TextOut(20,20,message);
```

The if statement obtains a pointer to the main window, casts it to a CMainFrame*, and asks the check box whether it is selected. Then the message is set appropriately.

Build the project and run it. As you select and deselect the check box, you should see the message change, as in Figure 9.19.

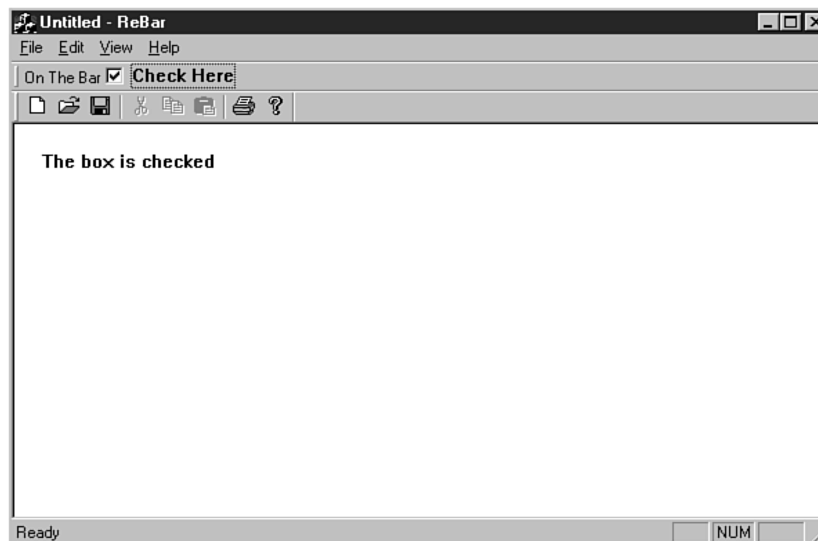


FIG. 9.19 Clicking the check box changes the view.

COMMON CONTROLS

In this chapter

The Progress Bar Control
The Slider Control
The Up-Down Control
The Image List Control
The List View Control
The Tree View Control
The Rich Edit Control
IP Address Control
The Date Picker Control
Month Calendar Control
Scrolling the View

As a Windows user, you're accustomed to seeing controls such as buttons, list boxes, menus, and edit boxes. As Windows developed, however, Microsoft noticed that developers routinely create other types of controls in their programs: toolbars, status bars, progress bars, tree views, and others. To make life easier for Windows programmers, Microsoft included these popular controls as part of the operating environment of Windows 95 (as well as later versions of Windows NT and then Windows 98). Now Windows programmers no longer need to create from scratch their own versions of these controls. This chapter introduces you to many of the 32-bit Windows common controls. The toolbar and status bar controls are covered in Chapter 9. This chapter's sample program is called Common. It demonstrates nine of the Windows 95 common controls: the progress bar, slider, up-down, list view, tree view, rich edit, IP address, date picker, and month calendar controls, all of which are shown in Figure 10.1. In the following sections, you learn the basics of creating and using these controls in your own applications.

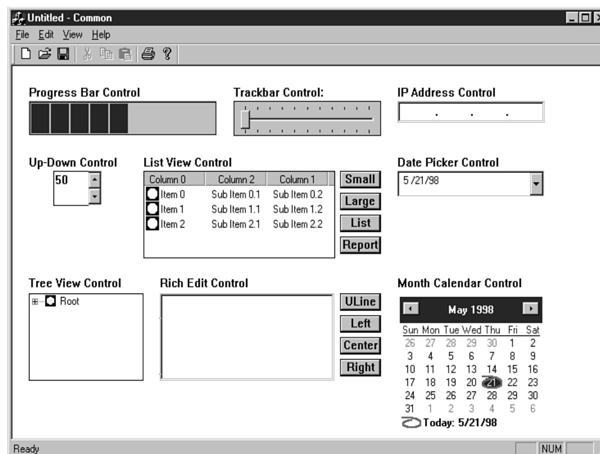


FIG. 10.1 The Common sample application demonstrates nine Windows 95 common controls.

To make Common, create a new project with AppWizard and name it **Common**. Choose a single-document interface (SDI) application in Step 1 and accept all the defaults until Step 6. Drop down the Base Class box and choose CScrollView from the list. This ensures that users can see all the controls in the view, even if they have to scroll to do so. Click Finish and then OK to complete the process.

The controls themselves are declared as data members of the view class. Double-click CCommonView in ClassView to edit the header file and add the lines in Listing 10.1 in the Attributes section. As you can see, the progress bar is an object of the CProgressCtrl class. It's discussed in the next section, and the other controls are discussed in later sections of this chapter.

Listing 10.1 CommonView.h—Declaring the Controls

```
protected:
    //Progress Bar
        CProgressCtrl m_progressBar;
    //Trackbar or Slider
        CSliderCtrl m_trackbar;
        BOOL m_timer;
    // Up-Down or Spinner
        CSpinButtonCtrl m_upDown;
        CEdit m_buddyEdit;
    // List View
        CListCtrl m_listView;
        CImageList m_smallImageList;
        CImageList m_largeImageList;
        CButton m_smallButton;
        CButton m_largeButton;
        CButton m_listButton;
        CButton m_reportButton;
    // Tree View
        CTreeCtrl m_treeView;
        CImageList m_treeImageList;
    // Rich Edit
        CRichEditCtrl m_richEdit;
        CButton m_boldButton;
        CButton m_leftButton;
        CButton m_centerButton;
        CButton m_rightButton;
```

```
// IP Address
        CIPAddressCtrl m_ipaddress;
// Date Picker
        CDateTimeCtrl m_date;
// Month Calendar
        CMonthCalCtrl m_month;
```

Expand the CCommonView class. Double-click CCommonView::OnDraw() in ClassView and replace the TODO comment with these lines:

```
pDC->TextOut(20, 22, "Progress Bar Control");
pDC->TextOut(270, 22, "Trackbar Control:");
pDC->TextOut(20, 102, "Up-Down Control");
pDC->TextOut(160, 102, "List View Control");
pDC->TextOut(20, 240, "Tree View Control");
pDC->TextOut(180, 240, "Rich Edit Control");
pDC->TextOut(470, 22, "IP Address Control");
pDC->TextOut(470, 102, "Date Picker Control");
pDC->TextOut(470, 240, "Month Calendar Control");
```

These label the controls that you will add to CCommonView in this chapter.

The Progress Bar Control

The common control that's probably easiest to use is the progress bar, which is nothing more than a rectangle that slowly fills in with colored blocks. The more colored blocks that are filled in, the closer the task is to being complete. When the progress bar is completely filled in, the task associated with the progress bar is also complete. You might use a progress bar to show the status of a sorting operation or to give the user visual feedback about a large file that's being loaded.

Creating the Progress Bar

Before you can use a progress bar, you must create it. Often in an MFC program, the controls are created as part of a dialog box. However, Common displays its controls in the application's main window, the view of this single-document interface (SDI) application. Documents and views are introduced in Chapter 4, "Documents and Views." All the controls are created in the view class OnCreate() function, which responds to the WM_CREATE Windows message. To set up this function, right-click CCommonView in ClassView and choose Add Windows Message Handler. Choose WM_CREATE from the list on the left and click Add and Edit. Add this line in place of the TODO comment:

```
CreateProgressBar();
```

Right-click CCommonView in ClassView again and this time choose Add Member Function. Enter void for the Function Type and enter CreateProgressBar() for the

Function Declaration.Leave the access as Public. Click OK to add the function; then add the code in Listing 10.2.

Listing 10.2 CommonView.cpp—CCommonView::CreateProgressBar()

```
void CCommonView::CreateProgressBar()
{
    m_progressBar.Create(WS_CHILD | WS_VISIBLE | WS_BORDER,
    CRect(20, 40, 250, 80), this, IDC_PROGRESSBAR);
    m_progressBar.SetRange(1, 100);
    m_progressBar.SetStep(10);
    m_progressBar.SetPos(50);
    m_timer = FALSE;
}
```

CreateProgressBar() first creates the progress bar control by calling the control's Create() function. This function's four arguments are the control's style flags, the control's size (as a CRect object), a pointer to the control's parent window, and the control's ID. The resource ID, IDC_PROGRESSBAR, is added by hand. To add resource symbols to your own applications, choose View, Resource Symbols and click the New button. Type in a resource ID Name, such as IDC_PROGRESSBAR, and accept the default Value Visual Studio provides.

The style constants are the same constants that you use for creating any type of window (a control is nothing more than a special kind of window, after all). In this case, you need at least the following:

- WS_CHILD Indicates that the control is a child window
- WS_VISIBLE Ensures that the user can see the control

The WS_BORDER is a nice addition because it adds a dark border around the control, setting it off from the rest of the window.

Initializing the Progress Bar

To initialize the control, CCommonView::CreateProgressBar() calls SetRange(), SetStep(), and SetPos(). Because the range and the step rate are related, a control with a range of 1–10 and a step rate of 1 works almost identically to a control with a range of 1–100 and a step rate of 10.

When this sample application starts, the progress bar is already half filled with colored blocks. (This is purely for aesthetic reasons. Usually a progress bar begins its life empty.) It's half full because CreateProgressBar() calls SetPos() with the value of 50, which is the midpoint of the control's range.

Manipulating the Progress Bar

Normally you update a progress bar as a long task moves toward completion. In this sample, you will fake it by using a timer. When the user clicks in the background of the view, start a timer that generates WM_TIMER messages periodically. Catch these messages and advance the progress bar. Here's what to do:

1. Open ClassWizard. Make sure that CCommonView is selected in the upper-right dropdown box.

2. Scroll most of the way through the list box on the right until you find WM_LBUTTONDOWN, the message generated when the user clicks on the view. Select it.

3. Click Add Function; then click Edit Code.

4. Edit OnLButtonDown()

so that it looks like this:

```
void CCommonView::OnLButtonDown(UINT nFlags, CPoint point)
{
    if (m_timer)
    {
        KillTimer(1);
        m_timer = FALSE;
    }
    else
    {
        SetTimer(1, 500, NULL);
        m_timer = TRUE;
    }
    CView::OnLButtonDown(nFlags, point);
}
```

This code enables users to turn the timer on or off with a click. The parameter of 500 in the SetTimer call is the number of milliseconds between WM_TIMER messages: This timer will send a message twice a second.

5. In case a timer is still going when the view closes, you should override OnDestroy() to kill the timer. Right-click CCommonView in ClassView yet again and choose Add Windows Message Handler. Select WM_DESTROY and click Add and Edit. Replace the TODO comment with this line:

```
KillTimer(1);
```

6. Now, catch the timer messages. Open ClassWizard and, as before, scroll through the list of messages in the far right list box. WM_TIMER is the second-to-last message in the alphabetic list, so drag the elevator all the way to the bottom and select WM_TIMER. Click Add Function and then click Edit Code. Replace the TODO comment with this line:

```
m_progressBar.StepIt();
```

The StepIt() function increments the progress bar control's value by the step rate, causing

new blocks to be displayed in the control as the control's value setting counts upward. When the control reaches its maximum, it automatically starts over.

Build Common and execute it to see the progress bar in action. Be sure to try stopping the timer as well as starting it.

The Slider Control

Many times in a program you might need the user to enter a value within a specific range. For this sort of task, you use MFC's CSliderCtrl class to create a slider (also called *trackbar*) control. For example, suppose you need the user to enter a percentage. In this case, you want the user to enter values only in the range of 0–100. Other values would be invalid and could cause problems in your program.

By using the slider control, you can force the user to enter a value in the specified range. Although the user can accidentally enter a wrong value (a value that doesn't accomplish what the user wants to do), there is no way to enter an invalid value (one that brings your program crashing down like a stone wall in an earthquake).

For a percentage, you create a slider control with a minimum value of 0 and a maximum value of 100. Moreover, to make the control easier to position, you might want to place tick marks at each setting that's a multiple of 10, providing 11 tick marks in all (including the one at 0). Common creates exactly this type of slider.

To use a slider, the user clicks the slider's slot. This moves the slider forward or backward, and often the selected value appears near the control. When a slider has the focus, the user can also control it with the Up and Down arrow keys and the Page Up and Page Down keys.

Creating the Trackbar

You are going to need a resource symbol for the trackbar control, so just as you did for the progress bar, choose View, Resource Symbols and click New. Enter IDC_TRACKBAR for the resource ID Name and accept the suggested Value. In CCommonView::OnCreate(), add a call to CreateTrackbar(). Then add the new member function as you added CreateProgressBar() and type in the code in Listing 10.3.

Listing 10.3 CommonView.cpp—CCommonView::CreateTrackBar()

```
void CCommonView::CreateTrackbar()
{
    m_trackbar.Create(WS_CHILD | WS_VISIBLE | WS_BORDER |
    TBS_AUTOTICKS | TBS_BOTH | TBS_HORZ,
    CRect(270, 40, 450, 80), this, IDC_TRACKBAR);
    m_trackbar.SetRange(0, 100, TRUE);
    m_trackbar.SetTicFreq(10);
```

```
        m_trackbar.SetLineSize(1);
        m_trackbar.SetPageSize(10);
}
```

As with the progress bar, the first step is to create the slider control by calling its Create() member function. This function's four arguments are the control's style flags, the control's size (as a CRect object), a pointer to the control's parent window, and the control's ID. The style constants include the same constants that you would use for creating any type of window, with the addition of special styles used with sliders. Table 10.1 lists these special styles.

Table 10.1 Slider Styles

Style	Description
TBS_AUTOTICKS	Enables the slider to automatically draw its tick marks
TBS_BOTH	Draws tick marks on both sides of the slider
TBS_BOTTOM	Draws tick marks on the bottom of a horizontal slider
TBS_ENABLESELRANGE	Enables a slider to display a subrange of values
TBS_HORZ	Draws the slider horizontally
TBS_LEFT	Draws tick marks on the left side of a vertical slider
TBS_NOTICKS	Draws a slider with no tick marks
TBS_RIGHT	Draws tick marks on the right side of a vertical slider
TBS_TOP	Draws tick marks on the top of a horizontal slider
TBS_VERT	Draws a vertical slider

Initializing the Trackbar

Usually, when you create a slider control, you want to set the control's range and tick frequency. If the user is going to use the control from the keyboard, you also need to set the control's line and page size. In Common, the program initializes the trackbar with calls to SetRange(), SetTicFreq(), SetLineSize(), and SetPageSize(), as you saw in Listing 10.3. The call to SetRange() sets the trackbar's minimum and maximum values to 0 and 100. The arguments are the minimum value, the maximum value, and a Boolean value indicating whether the slider should redraw itself after setting the range. Notice that the tick frequency and page size are then set to be the same. This isn't absolutely required, but it's a very good idea. Most people assume that the tick marks indicate the size of a page, and you will confuse your users if the tick marks are more or less than a page apart.

A number of other functions can change the size of your slider, the size of the thumb, the current selection, and more. You can find all the details in the online documentation.

Manipulating the Slider

A slider is really just a special scrollbar control. When the user moves the slider, the control generates WM_HSCROLL messages, which you will arrange to catch. Open ClassWizard, select the Message Maps tab, make sure CCommonView is selected in the upper-right box, and find WM_HSCROLL in the list on the right. Select it, click Add Function, and then click Edit Code. Type in the code in Listing 10.4.

Listing 10.4 CommonView.cpp—CCommonView::OnHScroll()

```
void CCommonView::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar)
{
    CSliderCtrl* slider = (CSliderCtrl*)pScrollBar;
    int position = slider->GetPos();
    char s[10];
    wsprintf(s, "%d ", position);
    CClientDC clientDC(this);
    clientDC.TextOut(390, 22, s);
    CScrollView::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

Looking at this code, you see that the control itself doesn’t display the current position as a number nearby; it’s the OnHScroll() function that displays the number. Here’s how it works:

1. OnHScroll()’s fourth parameter is a pointer to the scroll object that generated the WM_HSCROLL message.

2. The function first casts this pointer to a CSliderCtrl pointer; then it gets the current position of the trackbar’s slider by calling the CSliderCtrl member function GetPos().

3. After the program has the slider’s position, it converts the integer to a string and displays that string in the window with TextOut().

To learn how to make text appear onscreen, refer to Chapter 5, “Drawing on the Screen.” Before moving on to the next control, build Common and test it. Click around on the slider and watch the number change.

The Up-Down Control

The trackbar control isn’t the only way you can get a value in a predetermined range from the user. If you don’t need the trackbar for visual feedback, you can use an up-down control, which is little more than a couple of arrows that the user clicks to increase or decrease the control’s setting. Typically, an edit control next to the up-down control, called a *buddy edit* control or just a *buddy* control, displays the value to the user.

In the Common application, you can change the setting of the up-down control by clicking either of its arrows. When you do, the value in the attached edit box changes, indicating the updown control's current setting. After the control has the focus, you can also change its value by pressing your keyboard's Up and Down arrow keys.

Creating the Up-Down Control

Add another call to CCommonView::OnCreate(), this time calling it CreateUpDownCtrl(). Add the member function and the code in Listing 10.5. Also add resource symbols for IDC_BUDDYEDIT and IDC_UPDOWN.

Listing 10.5 CommonView.cpp—CCommonView::CreateUpDownCtrl()

```
void CCommonView::CreateUpDownCtrl()
{
    m_buddyEdit.Create(WS_CHILD | WS_VISIBLE | WS_BORDER,
    CRect(50, 120, 110, 160), this, IDC_BUDDYEDIT);
    m_upDown.Create(WS_CHILD | WS_VISIBLE | WS_BORDER |
    UDS_ALIGNRIGHT | UDS_SETBUDDYINT | UDS_ARROWKEYS,
    CRect(0, 0, 0, 0), this, IDC_UPDOWN);
    m_upDown.SetBuddy(&m_buddyEdit);
    m_upDown.SetRange(1, 100);
    m_upDown.SetPos(50);
}
```

The program creates the up-down control by first creating the associated buddy control to which the up-down control communicates its current value. In most cases, including this one, the buddy control is an edit box, created by calling the CEdit class's Create() member function. This function's four arguments are the control's style flags, the control's size, a pointer to the control's parent window, and the control's ID. If you recall the control declarations, m_buddyEdit is an object of the CEdit class.

Now that the program has created the buddy control, it can create the up-down control in much the same way, by calling the object's Create() member function. As you can probably guess by now, this function's four arguments are the control's style flags, the control's size, a pointer to the control's parent window, and the control's ID. As with most controls, the style constants include the same constants that you use for creating any type of window. The CSpinButtonCtrl class, of which m_upDown is an object, however, defines special styles to be used with up-down controls. Table 10.2 lists these special styles.

Table 10.2 Up-Down Control Styles

Styles	Description
UDS_ALIGNLEFT	Places the up-down control on the left edge of the buddy control
UDS_ALIGNRIGHT	Places the up-down control on the right edge of the buddy control
UDS_ARROWKEYS	Enables the user to change the control's values by using the keyboard's Up and Down arrow keys
UDS_AUTOBUDDY	Makes the previous window the buddy control
UDS_HORZ	Creates a horizontal up-down control
UDS_NOTHOUSANDS	Eliminates separators between each set of three digits
UDS_SETBUDDYINT	Displays the control's value in the buddy control
UDS_WRAP	Causes the control's value to wrap around to its minimum when the maximum is reached, and vice versa

This chapter's sample application establishes the up-down control with calls to SetBuddy(), SetRange(), and SetPos(). Thanks to the UDS_SETBUDDYINT flag passed to Create() and the call to the control's SetBuddy() member function, Common doesn't need to do anything else for the control's value to appear on the screen. The control automatically handles its buddy. Try building and testing now.

You might want up-down controls that move faster or slower than in this sample or that use hex numbers rather than base-10 numbers. Look at the member functions of this control in the online documentation, and you will see how to do that.

The Image List Control

Often you need to use images that are related in some way. For example, your application might have a toolbar with many command buttons, each of which uses a bitmap for its icon. In a case like this, it would be great to have some sort of program object that could not only hold the bitmaps but also organize them so that they can be accessed easily. That's exactly what an image list control does for you—it stores a list of related images. You can use the images any way that you see fit in your program. Several common controls rely on image lists. These controls include the following:

- List view controls
- Tree view controls
- Property pages
- Toolbars

You will undoubtedly come up with many other uses for image lists. You might, for example, have an animation sequence that you'd like to display in a window. An image list is the perfect storage place for the frames that make up an animation, because you can easily access any frame just by using an index.

If the word *index* makes you think of arrays, you're beginning to understand how an image list stores images. An image list is ver y similar to an array that holds pictures rather than integers or floating-point numbers. Just as with an array, you initialize each "element" of an image list and thereafter can access any part of the "array" by using an index. You won't, however, see an image list control in your running application in the same way that you can see a status bar or a progress bar control. This is because (again, similar to an array) an image list is only a storage structure for pictures.

Creating the Image List

In the Common Controls App application, image lists are used with the list view and tree view controls, so the image lists for the controls are created in the CreateListView() and CreateTreeView() local member functions and are called from CCommonView::OnCreate(). Just as with the other controls, add calls to these functions to OnCreate() and then add the functions to the class. You will see the full code for those functions shortly, but because they are long, this section presents the parts that are relevant to the image list. A list view uses two image lists: one for small images and the other for large ones. The member variables for these lists have already been added to the class, so start coding CreateListView() with a call to each list's Create() member function, like this:

```
m_smallImageList.Create(16, 16, FALSE, 1, 0);
m_largeImageList.Create(32, 32, FALSE, 1, 0);
```

The Create() function's five arguments are

- The width of the pictures in the control
- The height of the pictures
- A Boolean value indicating whether the images contain a mask
- The number of images initially in the list
- The number of images by which the list can dynamically grow

This last value is 0 to indicate that the list isn't allowed to grow during runtime. The Create() function is overloaded in the CImageList class so that you can create image lists in various ways. You can find the other versions of Create() in your Visual C++ online documentation.

Initializing the Image List

After you create an image list, you will want to add images to it. After all, an empty image list isn't of much use. The easiest way to add the images is to include the images as part of your application's resource file and load them from there. Add these four lines to CreateListView() to fill each list with images:

```
HICON hIcon = ::LoadIcon (AfxGetResourceHandle(),
MAKEINTRESOURCE(IDI_ICON1));
m_smallImageList.Add(hIcon);
hIcon = ::LoadIcon (AfxGetResourceHandle(),
MAKEINTRESOURCE(IDI_ICON2));
m_largeImageList.Add(hIcon);
```

Here the program first gets a handle to the icon. Then it adds the icon to the image list by calling the image list's Add() member function. (In this case, the list includes only one icon. In other applications, you might have a list of large icons for folders, text files, and so on, as well as another list of small icons for the same purposes.) To create the first icon, choose Insert, Resource and double-click Icon. Then edit the new blank icon in the Resource Editor. (It will automatically be called IDI_ICON1.) Click the New Device Image toolbar button next to the drop-down box that says Standard (32´32) and choose Small (16´16) on the dialog that appears; click OK. You can spend a long time making a beautiful icon or just quickly fill in the whole grid with black and then put a white circle on it with the Ellipse tool. Add another icon, IDI_ICON2, and leave it as 32´32. Draw a similar symbol on this icon. You can use many member functions to manipulate an object of the CImageList class, adjusting colors, removing images, and much more. The online documentation provides more details on these member functions.

You can write the first few lines of CreateTreeView() now. It uses one image list that starts with three images. Here's the code to add:

```
m_treeImageList.Create(13, 13, FALSE, 3, 0);
HICON hIcon = ::LoadIcon(AfxGetResourceHandle(),
MAKEINTRESOURCE(IDI_ICON3));
m_treeImageList.Add(hIcon);
hIcon = ::LoadIcon(AfxGetResourceHandle(),
MAKEINTRESOURCE(IDI_ICON4));
m_treeImageList.Add(hIcon);
hIcon = ::LoadIcon(AfxGetResourceHandle(),
MAKEINTRESOURCE(IDI_ICON5));
m_treeImageList.Add(hIcon);
```

Create IDI_ICON3, IDI_ICON4, and IDI_ICON5 the same way you did the first two icons. All three are 32´32. Draw circles as before. If you leave the background the same murky green you started with, rather than fill it with black, the circles will appear on a transparent background— a nice effect.

The List View Control

A list view control simplifies the job of building an application that works with lists of objects and organizes those objects in such a way that the program's user can easily determine each object's attributes. For example, consider a group of files on a disk. Each file is a separate object associated with a number of attributes, including the file's name, size, and the most recent modification date. When you explore a folder, you see files either as icons in a window or as a table of entries, each entry showing the attributes associated with the files.

You have full control over the way that the file objects are displayed, including which attributes are shown and which are unlisted. The common controls include something called a *list view control*, so you can organize lists in exactly the same way. If you'd like to see an example of a full-fledged list view control, open the Windows

Explorer (see Figure 10.3). The right side of the window shows how the list view control can organize objects in a window. (The left side of the window contains a tree view control, which you will learn about later in this chapter in the section titled "The Tree View Control.") In the figure, the list view is currently set to the report view, in which each object in the list receives its own line, showing not only the object's name but also the attributes associated with that object.

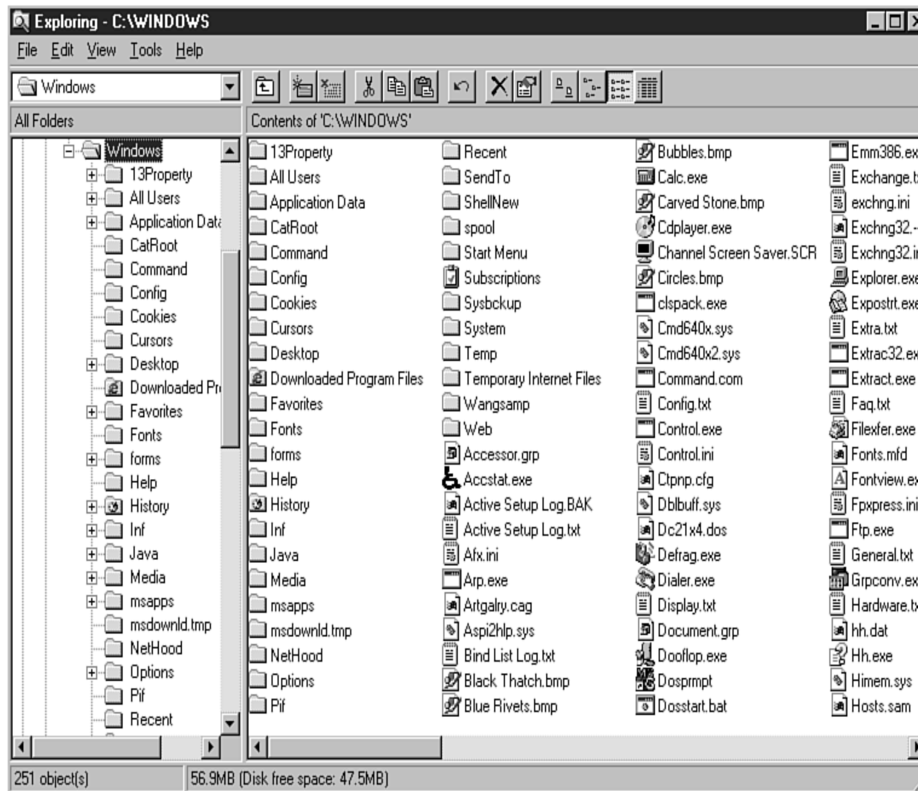


FIG. 10.3 Windows Explorer uses a list view control to organize file information

The user can change the way objects are organized in a list view control. Figure 10.4, for example, shows the list view portion of the Explorer set to the large-icon setting, and Figure 10.5 shows the small-icon setting, which enables the user to see more objects (in this case, files) in the window. With a list view control, the user can edit the names of objects in the list and in the report view can sort objects, based on data displayed in a particular column.

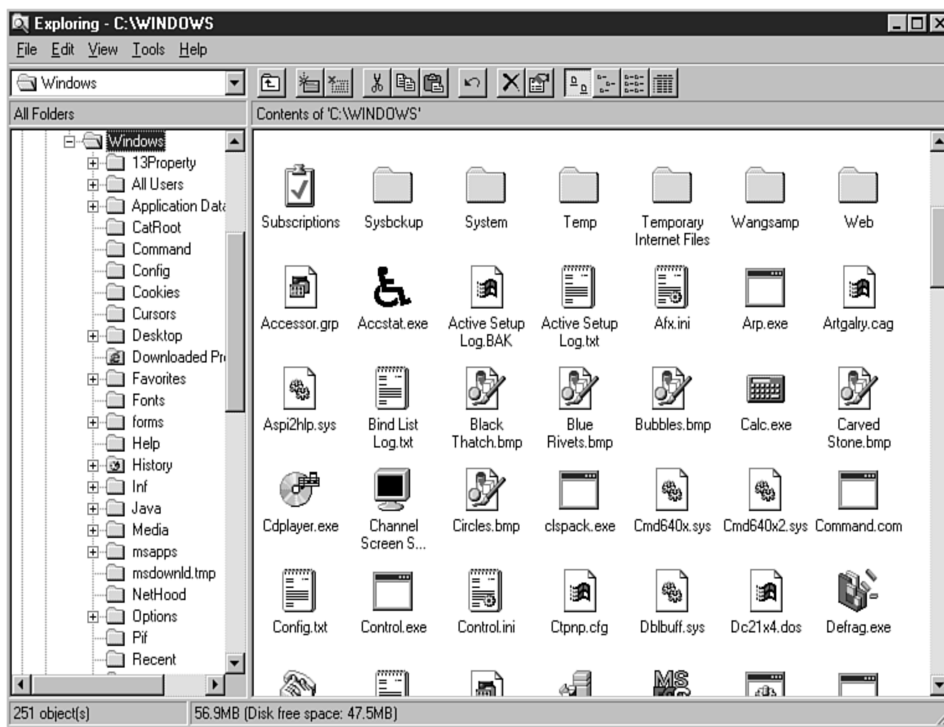


FIG. 10.4 Here's Explorer's list view control set to large icons.

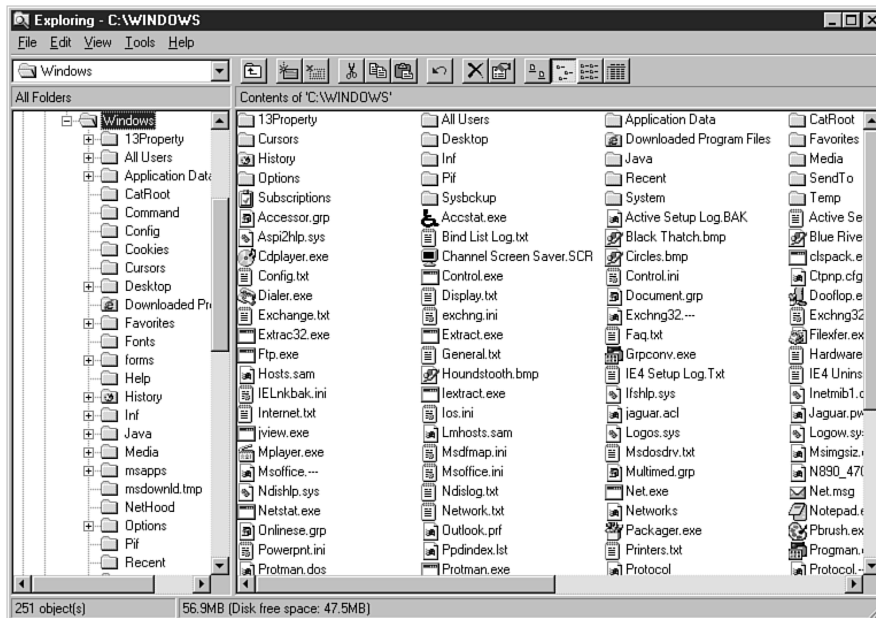


FIG. 10.5 Here's Explorer's list view control set to small icons.

Common will also sport a list view control, although not as fancy as Explorer's. You will add a list view and some buttons to switch between the small-icon, large-icon, list, and report views.

Creating the List View

How does all this happen? Well, it does require more work than the progress bar, trackbar, or up-down controls (it could hardly take less). You will write the rest of CreateListView(), which performs the following tasks:

1. Creates and fills the image list controls
2. Creates the list view control itself
3. Associates the image lists with the list view
4. Creates the columns
5. Sets up the columns
6. Creates the items
7. Sets up the items
8. Creates the buttons

After creating the image lists, CreateListView() goes on to create the list view control by calling the class's Create() member function, as usual. Add these lines to CreateListView():

```
// Create the List View control.
m_listView.Create(WS_VISIBLE | WS_CHILD | WS_BORDER |
LVS_REPORT | LVS_NOSORTHEADER | LVS_EDITLABELS,
CRect(160, 120, 394, 220), this, IDC_LISTVIEW);
```

The CListCtrl class, of which m_listView is an object, defines special styles to be used with list view controls. Table 10.3 lists these special styles and their descriptions.

Table 10.3 List View Styles

Style	Description
LVS_ALIGNLEFT	Left-aligns items in the large-icon and small-icon views
LVS_ALIGNTOP	Top-aligns items in the large-icon and small-icon views
LVS_AUTOARRANGE	Automatically arranges items in the large-icon and small-icon views
LVS_EDITLABELS	Enables the user to edit item labels
LVS_ICON	Sets the control to the large-icon view
LVS_LIST	Sets the control to the list view
LVS_NOCOLUMNHEADER	Shows no column headers in report view
LVS_NOITEMDATA	Stores only the state of each item

LVS_NOLABELWRAP	Disallows multiple-line item labels
LVS_NOSCROLL	Turns off scrolling
LVS_NOSORTHEADER	Turns off the button appearance of column headers
LVS_OWNERDRAWFIXED	Enables owner-drawn items in report view
LVS_REPORT	Sets the control to the report view
LVS_SHAREIMAGELISTS	Prevents the control from destroying its image lists when the control no longer needs them
LVS_SINGLESEL	Disallows multiple selection of items
LVS_SMALLICON	Sets the control to the small-icon view
LVS_SORTASCENDING	Sorts items in ascending order
LVS_SORTDESCENDING	Sorts items in descending order

The third task in CreateListView() is to associate the control with its image lists with two calls to SetImageList(). Add these lines to CreateListView():

```
m_listView.SetImageList(&m_smallImageList, LVSIL_SMALL);
m_listView.SetImageList(&m_largeImageList, LVSIL_NORMAL);
```

This function takes two parameters: a pointer to the image list and a flag indicating how the list is to be used. Three constants are defined for this flag: LVSIL_SMALL (which indicates that the list contains small icons), LVSIL_NORMAL (large icons), and LVSIL_STATE (state images). The SetImageList() function returns a pointer to the previously set image list, if any.

Creating the List View's Columns

The fourth task is to create the columns for the control's report view. You need one main column for the item itself and one column for each sub-item associated with an item. For example, in Explorer's list view, the main column holds file and folder names. Each additional column holds the sub-items for each item, such as the file's size, type, and modification date. To create a column, you must first declare a LV_COLUMN structure. You use this structure to pass information to and from the system. After you add the column to the control with InsertColumn(), you can use the structure to create and insert another column. Listing 10.6 shows the LV_COLUMN structure.

Listing 10.6 The LV_COLUMN Structure, Defined by MFC

```
typedef struct _LV_COLUMN
{
UINT mask; // Flags indicating valid fields
int fmt; // Column alignment
int cx; // Column width
LPSTR pszText; // Address of string buffer
```

```
int cchTextMax; // Size of the buffer
int iSubItem; // Subitem index for this column
} LV_COLUMN;
```

The mask member of the structure tells the system which members of the structure to use and

which to ignore. The flags you can use are

- LVCF_FMT fmt is valid.
- LVCF_SUBITEM iSubItem is valid.
- LVCF_TEXT pszText is valid.
- LVCF_WIDTH cx is valid.

The fmt member denotes the column's alignment and can be LVCFMT_CENTER, LVCFMT_LEFT, or LVCFMT_RIGHT. The alignment determines how the column's label and items are positioned in the column.

The cx field specifies the width of each column, whereas pszText is the address of a string buffer. When you're using the structure to create a column (you also can use this structure to obtain information about a column), this string buffer contains the column's label. The cchTextMax member denotes the size of the string buffer and is valid only when retrieving information about a column.

CreateListView() creates a temporary LV_COLUMN structure, sets the elements, and then inserts it into the list view as column 0, the main column. This process is repeated for the other two columns. Add these lines to CreateListView():

```
// Create the columns.
LV_COLUMN lvColumn;
lvColumn.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT | LVCF_SUBITEM;
lvColumn.fmt = LVCFMT_CENTER;
lvColumn.cx = 75;
lvColumn.iSubItem = 0;
lvColumn.pszText = “Column 0”;
m_listView.InsertColumn(0, &lvColumn);
lvColumn.iSubItem = 1;
lvColumn.pszText = “Column 1”;
m_listView.InsertColumn(1, &lvColumn);
lvColumn.iSubItem = 2;
lvColumn.pszText = “Column 2”;
m_listView.InsertColumn(1, &lvColumn);
```

Creating the List View's Items

The fifth task in CreateListView() is to create the items that will be listed in the columns when the control is in its report view. Creating items is not unlike creating columns. As with columns, Visual C++ defines a structure that you must initialize and pass to the function that creates the items. This structure is called LV_ITEM and is defined as shown in Listing 10.7.

Listing 10.7 The LV_ITEM Structure, Defined by MFC

```
typedef struct _LV_ITEM
{
    UINT mask; // Flags indicating valid fields
    int iItem; // Item index
    int iSubItem; // Sub-item index
    UINT state; // Item's current state
    UINT stateMask; // Valid item states.
    LPSTR pszText; // Address of string buffer
    int cchTextMax; // Size of string buffer
    int iImage; // Image index for this item
    LPARAM lParam; // Additional information as a 32-bit value
} LV_ITEM;
```

In the LV_ITEM structure, the mask member specifies the other members of the structure that are valid. The flags you can use are

- LVIF_IMAGE iImage is valid.
- LVIF_PARAM lParam is valid.
- LVIF_STATE state is valid.
- LVIF_TEXT pszText is valid.

The iItem member is the index of the item, which you can think of as the row number in report view (although the items' position can change when they're sorted). Each item has a unique index. The iSubItem member is the index of the sub-item, if this structure is defining a sub-item. You can think of this value as the number of the column in which the item will appear. For example, if you're defining the main item (the first column), this value should be 0. The state and stateMask members hold the item's current state and its valid states, which can be one or more of the following:

- LVIS_CUT The item is selected for cut and paste.
- LVIS_DROPHILITED The item is a highlighted drop target.
- LVIS_FOCUSED The item has the focus.
- LVIS_SELECTED The item is selected.

The pszText member is the address of a string buffer. When you use the LV_ITEM structure to create an item, the string buffer contains the item's text. When you are obtaining information about the item, pszText is the buffer where the information will be stored, and cchTextMax is the size of the buffer. If pszText is set to LPSTR_TEXTCALLBACK, the item uses the callback mechanism. Finally, the iImage member is the index of the item's icon in the small-icon and large-icon image lists. If set to I_IMAGECALLBACK, the iImage member indicates that the item uses the callback mechanism.

CreateListView() creates a temporary LV_ITEM structure, sets the elements, and then inserts it into the list view as item 0. Two calls to SetItemText() add sub-items to this item so that each column has some text in it, and the whole process is repeated for two other items. Add these lines:

```
// Create the items.
    LV_ITEM lvItem;
    lvItem.mask = LVIF_TEXT | LVIF_IMAGE | LVIF_STATE;
    lvItem.state = 0;
    lvItem.stateMask = 0;
    lvItem.iImage = 0;
    lvItem.iItem = 0;
    lvItem.iSubItem = 0;
    lvItem.pszText = "Item 0";
    m_listView.InsertItem(&lvItem);
    m_listView.SetItemText(0, 1, "Sub Item 0.1");
    m_listView.SetItemText(0, 2, "Sub Item 0.2");
    lvItem.iItem = 1;
    lvItem.iSubItem = 0;
    lvItem.pszText = "Item 1";
    m_listView.InsertItem(&lvItem);
    m_listView.SetItemText(1, 1, "Sub Item 1.1");
    m_listView.SetItemText(1, 2, "Sub Item 1.2");
    lvItem.iItem = 2;
    lvItem.iSubItem = 0;
    lvItem.pszText = "Item 2";
    m_listView.InsertItem(&lvItem);
    m_listView.SetItemText(2, 1, "Sub Item 2.1");
    m_listView.SetItemText(2, 2, "Sub Item 2.2");
```

Now you have created a list view with three columns and three items. Normally the values wouldn't be hard-coded, as this was, but instead would be filled in with values calculated by the program.

Manipulating the List View

You can set a list view control to four different types of views: small icon, large icon, list, and report. In Explorer, for example, the toolbar features buttons that you can click to change the view, or you can select the view from the View menu. Although Common doesn't have a snazzy toolbar like Explorer, it will include four buttons (labeled Small, Large, List, and Report) that you can click to change the view. Those buttons are created as the sixth step in CreateListView(). Add these lines to complete the function:

```
// Create the view-control buttons.
    m_smallButton.Create("Small", WS_VISIBLE | WS_CHILD | WS_BORDER,
    CRect(400, 120, 450, 140), this, IDC_LISTVIEW_SMALL);
    m_largeButton.Create("Large", WS_VISIBLE | WS_CHILD | WS_BORDER,
    CRect(400, 145, 450, 165), this, IDC_LISTVIEW_LARGE);
    m_listButton.Create("List", WS_VISIBLE | WS_CHILD | WS_BORDER,
    CRect(400, 170, 450, 190), this, IDC_LISTVIEW_LIST);
    m_reportButton.Create("Report", WS_VISIBLE | WS_CHILD | WS_BORDER,
    CRect(400, 195, 450, 215), this, IDC_LISTVIEW_REPORT);
```

Edit the message map in CommonView.h to declare the handlers for each of these buttons so that it looks like this:

```
    // Generated message map functions
    protected:
    //{{AFX_MSG(CCommonView)
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
        afx_msg void OnDestroy();
        afx_msg void OnTimer(UINT nIDEvent);
        afx_msg void OnHScroll(UINT nSBCode, UINT nPos, CScrollBar*
        pScrollBar);
        //}}AFX_MSG
        afx_msg void OnSmall();
        afx_msg void OnLarge();
        afx_msg void OnList();
```

```
        afx_msg void OnReport();
        DECLARE_MESSAGE_MAP()
};
```

Edit the message map in CommonView.cpp to associate the messages with the functions:

```
BEGIN_MESSAGE_MAP(CCommonView, CScrollView)
        //{{AFX_MSG_MAP(CCommonView)
        ON_WM_CREATE()
        ON_WM_LBUTTONDOWN()
        ON_WM_DESTROY()
        ON_WM_TIMER()
        ON_WM_HSCROLL()
        //}}AFX_MSG_MAP
        ON_COMMAND(IDC_LISTVIEW_SMALL, OnSmall)
        ON_COMMAND(IDC_LISTVIEW_LARGE, OnLarge)
        ON_COMMAND(IDC_LISTVIEW_LIST, OnList)
        ON_COMMAND(IDC_LISTVIEW_REPORT, OnReport)
        // Standard printing commands
        ON_COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_DIRECT, CScrollView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_PREVIEW,
        CScrollView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

Choose View, Resource Symbols and click New to add new IDs for each constant referred to in this new code:

- IDC_LISTVIEW
- IDC_LISTVIEW_SMALL
- IDC_LISTVIEW_LARGE
- IDC_LISTVIEW_LIST
- IDC_LISTVIEW_REPORT

The four handlers will each call SetWindowLong(), which sets a window's attribute. Its arguments are the window's handle, a flag that specifies the value to be changed, and the new value. For example, passing GWL_STYLE as the second value means that the window's style should be changed to the style given in the third argument.

Changing the list view control's style (for example, to LVS_SMALLICON) changes the type of view that it displays. With that in mind, add the four handler functions to the bottom of CommonView.cpp:

```
void CCommonView::OnSmall()
{
    SetWindowLong(m_listView.m_hWnd, GWL_STYLE,
    WS_VISIBLE | WS_CHILD | WS_BORDER |
    LVS_SMALLICON | LVS_EDITLABELS);
}
void CCommonView::OnLarge()
{
    SetWindowLong(m_listView.m_hWnd, GWL_STYLE,
    WS_VISIBLE | WS_CHILD | WS_BORDER |
    LVS_ICON | LVS_EDITLABELS);
}
void CCommonView::OnList()
{
    SetWindowLong(m_listView.m_hWnd, GWL_STYLE,
    WS_VISIBLE | WS_CHILD | WS_BORDER |
    LVS_LIST | LVS_EDITLABELS);
}
void CCommonView::OnReport()
{
    SetWindowLong(m_listView.m_hWnd, GWL_STYLE,
    WS_VISIBLE | WS_CHILD | WS_BORDER |
    LVS_REPORT | LVS_EDITLABELS);
}
```

In addition to changing the view, you can program a number of other features for your list view controls. When the user does something with the control, Windows sends a WM_NOTIFY message to the parent window. The most common notifications sent by a list view control are thefollowing:

- LVN_COLUMNCLICK Indicates that the user clicked a column header
- LVN_BEGINLABELEDIT Indicates that the user is about to edit an item's label
- LVN_ENDLABELEDIT Indicates that the user is ending the label-editing process

Why not have Common allow editing of the first column in this list view? You start by overriding the virtual function OnNotify() that was inherited by CCommonView from CScrollView. Right-click CCommonView in ClassView and choose Add Virtual Function. Select OnNotify() from the list on the left and click Add and Edit; then add these lines of code at the beginning of the function, replacing the TODO comment:

```
LV_DISPINFO* lv_dispInfo = (LV_DISPINFO*) lParam;
if (lv_dispInfo->hdr.code == LVN_BEGINLABELEDIT)
{
    CEdit* pEdit = m_listView.GetEditControl();
    // Manipulate edit control here.
}
else if (lv_dispInfo->hdr.code == LVN_ENDLABELEDIT)
{
    if ((lv_dispInfo->item.pszText != NULL) &&
    (lv_dispInfo->item.iItem != -1))
    {
            m_listView.SetItemText(lv_dispInfo->item.iItem,
            0, lv_dispInfo->item.pszText);
    }
}
```

The three parameters received by OnNotify() are the message's WPARAM and LPARAM values and a pointer to a result code. In the case of a WM_NOTIFY message coming from a list view control, the WPARAM is the list view control's ID. If the WM_NOTIFY message is the LVN_BEGINLABELEDIT or LVN_ENDLABELEDIT notification, the LPARAM is a pointer to an LV_DISPINFO structure, which itself contains NMHDR and LV_ITEM structures. You use the information in these structures to manipulate the item that the user is trying to edit.

If the notification is LVN_BEGINLABELEDIT, your program can do whatever pre-editing initialization it needs to do, usually by calling GetEditControl() and then working with the pointer returned to you. This sample application shows you only how to get that pointer. When handling label editing, the other notification to watch out for is LVN_ENDLABELEDIT, which means that the user has finished editing the label, by either typing the new label or canceling the editing process. If the user has canceled the process, the LV_DISPINFO structure's item.pszText member will be NULL, or the item.iItem member will be –1. In this case, you need do nothing more than ignore the notification. If, however, the user completed the editing process, the program must copy the new label to the item's text, which OnNotify() does with a call to SetItemText(). The CListCtrl object's SetItemText() member function requires three arguments: the item index, the sub-item index, and the new text.

At this point you can build Common again and test it. Click each of the four buttons to change the view style. Also, try editing one of the labels in the first column of the list view. Figure 10.1 already showed you the report view for this list view. Figure 10.6 shows the application's list view control displaying small icons, and Figure 10.7 shows

You can do a lot of other things with a list view control. A little time invested in exploring and experimenting can save you a lot of time writing your user interface.

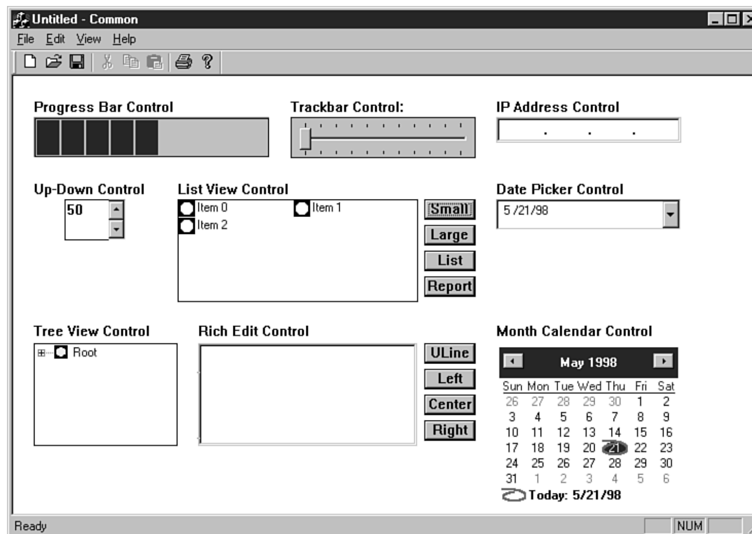


FIG. 10.6 Here's the sample application's list view control set to small icons.

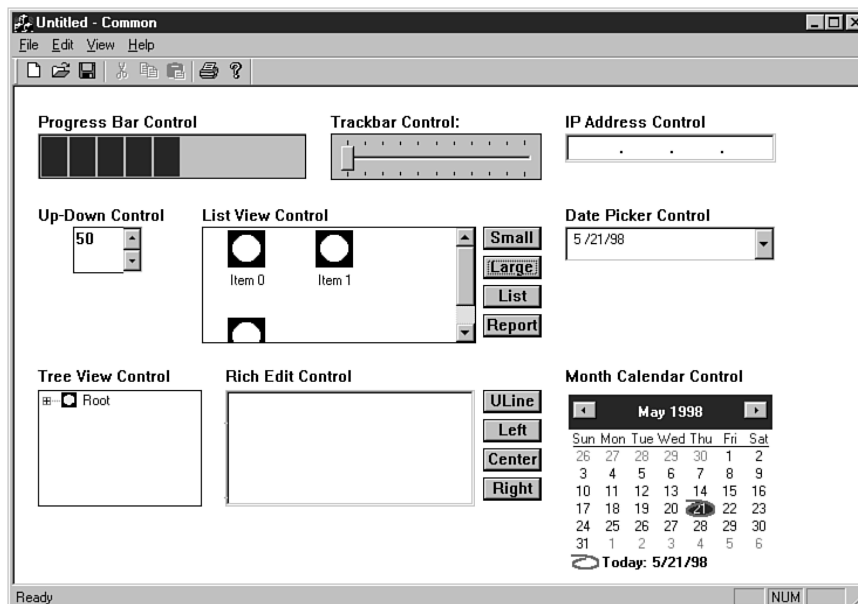


FIG. 10.7 Here's the sample application's list view control set to large icons.

The Tree View Control

In the preceding section, you learned how to use the list view control to organize the display of many items in a window. The list view control enables you to display items both as objects in a window and objects in a report organized into columns. Often, however, the data you'd like to organize for your application's user is best placed in a hierarchical view. That is, elements of the data are shown as they relate to one other. A good example of a hierarchical display is the directory tree used by Windows to display directories and the files that they contain. MFC provides this functionality in the CTreeCtrl class. This versatile control displays data in various ways, all the while retaining the hierarchical relationship between the data objects in the view.

If you'd like to see an example of a tree view control, revisit Windows Explorer (see Figure 10.8). The left side of the window shows how the tree view control organizes objects in a window. (The right side of the window contains a list view control, which you learned about in the preceding section). In the figure, the tree view displays not only the storage devices on the computer but also the directories and files stored on those devices. The tree clearly shows the hierarchical relationship between the devices, directories, and files, and it enables the user to open and close branches on the tree to explore different levels.

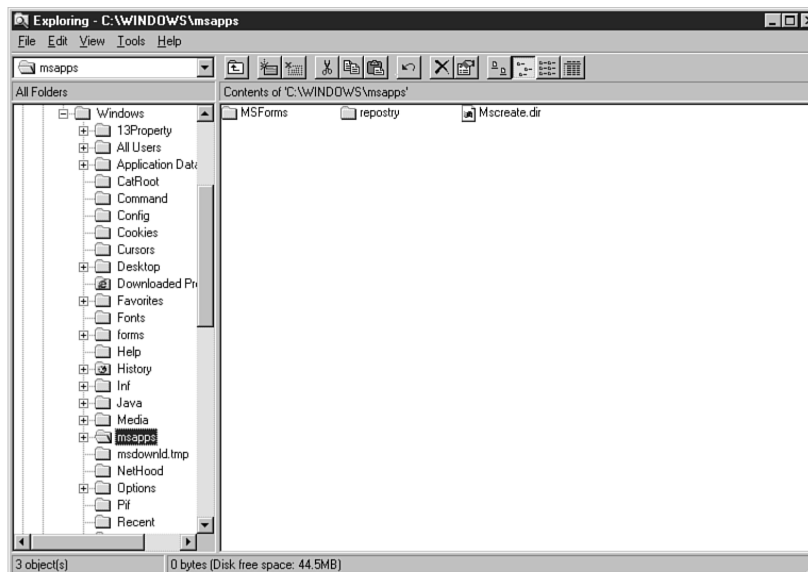


FIG. 10.8 A tree view control displays a hierarchical relationship between items

Creating the Tree View

Tree views are a little simpler than list views. You will write the rest of CreateTreeView(), which performs the following tasks:

1. Creates an image list

2. Creates the tree view itself

3. Associates the image list with the list view

4. Creates the root item

5. Creates child items

Creating the image list, creating the tree control, and associating the control with the image list are very similar to the steps completed for the image list. You've already written the code to create the image list, so add these lines to CreateTreeView():

```
// Create the Tree View control.
    m_treeView.Create(WS_VISIBLE | WS_CHILD | WS_BORDER |
        TVS_HASLINES | TVS_LINESATROOT | TVS_HASBUTTONS |
        TVS_EDITLABELS, CRect(20, 260, 160, 360), this,
        IDC_TREEVIEW);
    m_treeView.SetImageList(&m_treeImageList, TVSIL_NORMAL);
```

(Remember to add a resource ID for IDC_TREEVIEW.) The CTreeCtrl class, of which m_treeView is an object, defines special styles to be used with tree view controls. Table 10.4 lists these special styles.

Table 10.4 Tree View Control Styles

Style	Description
TVS_DISABLEDRAGDROP	Disables drag-and-drop operations
TVS_EDITLABELS	Enables the user to edit labels
TVS_HASBUTTONS	Gives each parent item a button
TVS_HASLINES	Adds lines between items in the tree
TVS_LINESATROOT	Adds a line between the root and child items
TVS_SHOWSELALWAYS	Forces a selected item to stay selected when losing focus
TVS_NOTOOLTIPS	Suppresses ToolTips for the tree items
TVS_SINGLEEXPAND	Expands or collapses tree items with a single click rather than a double click

Creating the Tree View's Items

Creating items for a tree view control is much like creating items for a list view control. As with the list view, Visual C++ defines a structure that you must initialize and pass to the function that creates the items. This structure is called TVITEM and is defined in Listing 10.8.

Listing 10.8 The TVITEM Structure, Defined by MFC

```
typedef struct _TVITEM
{
    UINT mask;
    HTREEITEM hItem;
    UINT state;
    UINT stateMask;
    LPSTR pszText;
    int cchTextMax;
    int iImage;
    int iSelectedImage;
    int cChildren;
    LPARAM lParam;
} TV_ITEM;
```

In the TVITEM structure, the mask member specifies the other structure members that are valid.

The flags you can use are as follows:

- TVIF_CHILDREN cChildren is valid.
- TVIF_HANDLE hItem is valid.
- TVIF_IMAGE iImage is valid.
- TVIF_PARAM lParam is valid.
- TVIF_SELECTEDIMAGE iSelectedImage is valid.
- TVIF_STATE state and stateMask are valid.
- TVIF_TEXT pszText and cchTextMax are valid.

The hItem member is the handle of the item, whereas the state and stateMask members hold the item's current state and its valid states, which can be one or more of

TVIS_BOLD, TVIS_CUT, TVIS_DROPHILITED, TVIS_EXPANDED, TVIS_EXPANDEDONCE, TVIS_FOCUSED, TVIS_OVERLAYMASK, TVIS_SELECTED, TVIS_STATEIMAGEMASK, and TVIS_USERMASK.

The pszText member is the address of a string buffer. When using the TVITEM structure to create an item, the string buffer contains the item's text. When obtaining information about the item, pszText is the buffer where the information will be stored,

and cchTextMax is the size of the buffer. If pszText is set to PSTR_TEXTCALLBACK, the item uses the callback mechanism.

Finally, the iImage member is the index of the item's icon in the image list. If set to I_IMAGECALLBACK, the iImage member indicates that the item uses the callback mechanism. The iSelectedImage member is the index of the icon in the image list that represents the item when the item is selected. As with iImage, if this member is set to I_IMAGECALLBACK, the iSelectedImage member indicates that the item uses the callback mechanism. Finally, cChildren specifies whether there are child items associated with the item. In addition to the TVITEM structure, you must initialize a TVINSERTSTRUCT structure that holds information about how to insert the new structure into the tree view control. That structure is declared in Listing 10.9.

Listing 10.9 The TVINSERTSTRUCT Structure, Defined by MFC

```
typedef struct tagTVINSERTSTRUCT {
    HTREEITEM hParent;
    HTREEITEM hInsertAfter;
#if (_WIN32_IE >= 0x0400)
    Union
    {
        TVITEMEX itemex;
        TVITEM item;
    } DUMMYUNIONNAME;
#else
    TVITEM item;
#endif
} TVINSERTSTRUCT, FAR *LPTVINSERTSTRUCT;
```

In this structure, hParent is the handle to the parent tree-view item. A value of NULL or TVI_ROOT specifies that the item should be placed at the root of the tree. The hInsertAfter member specifies the handle of the item after which this new item should be inserted. It can also be one of the flags TVI_FIRST (beginning of the list), TVI_LAST (end of the list), or TVI_SORT (alphabetical order). Finally, the item member is the TVITEM structure containing information about the item to be inserted into the tree.

Common first initializes the TVITEM structure for the root item (the first item in the tree). Add these lines:

```
// Create the root item.
    TVITEM tvItem;
    tvItem.mask =
    TVIF_TEXT | TVIF_IMAGE | TVIF_SELECTEDIMAGE;
```

```
tvItem.pszText = "Root";
tvItem.cchTextMax = 4;
tvItem.iImage = 0;
tvItem.iSelectedImage = 0;
TVINSERTSTRUCT tvInsert;
tvInsert.hParent = TVI_ROOT;
tvInsert.hInsertAfter = TVI_FIRST;
tvInsert.item = tvItem;
HTREEITEM hRoot = m_treeView.InsertItem(&tvInsert);
```

The CTreeCtrl member function InsertItem() inserts the item into the tree view control. Its single argument is the address of the TVINSERTSTRUCT structure.

CreateTreeView() then inserts the remaining items into the tree view control. Add these lines to insert some hard-coded sample items into the tree view:

```
// Create the first child item.
    tvItem.pszText = "Child Item 1";
    tvItem.cchTextMax = 12;
    tvItem.iImage = 1;
    tvItem.iSelectedImage = 1;
    tvInsert.hParent = hRoot;
    tvInsert.hInsertAfter = TVI_FIRST;
    tvInsert.item = tvItem;
    HTREEITEM hChildItem = m_treeView.InsertItem(&tvInsert);

// Create a child of the first child item.
    tvItem.pszText = "Child Item 2";
    tvItem.cchTextMax = 12;
    tvItem.iImage = 2;
    tvItem.iSelectedImage = 2;
    tvInsert.hParent = hChildItem;
    tvInsert.hInsertAfter = TVI_FIRST;
    tvInsert.item = tvItem;
    m_treeView.InsertItem(&tvInsert);
```

```
// Create another child of the root item.
        tvItem.pszText = "Child Item 3";
        tvItem.cchTextMax = 12;
        tvItem.iImage = 1;
        tvItem.iSelectedImage = 1;
        tvInsert.hParent = hRoot;
        tvInsert.hInsertAfter = TVI_LAST;
        tvInsert.item = tvItem;
        m_treeView.InsertItem(&tvInsert);
```

Manipulating the Tree View

Just as with the list view control, you can edit the labels of the items in Common's tree view.Also, like the list view control, this process works because the tree view sends WM_NOTIFY messages that trigger a call to the program's OnNotify() function. OnNotify() handles the tree-view notifications in almost exactly the same way as the list-view notifications. The only difference is in the names of the structures used. Add these lines to OnNotify() before the return statement:

```
TV_DISPINFO* tv_dispInfo = (TV_DISPINFO*) lParam;
        if (tv_dispInfo->hdr.code == TVN_BEGINLABELEDIT)
        {
                CEdit* pEdit = m_treeView.GetEditControl();
                // Manipulate edit control here.
        }
        else if (tv_dispInfo->hdr.code == TVN_ENDLABELEDIT)
        {
                if (tv_dispInfo->item.pszText != NULL)
                {
                        m_treeView.SetItemText(tv_dispInfo->item.hItem,
                        tv_dispInfo->item.pszText);
                }
        }
```

The tree view control sends a number of other notification messages, including TVN_BEGINDRAG, TVN_BEGINLABELEDIT, TVN_BEGINRDRAG, TVN_DELETEITEM, TVN_ENDLABELEDIT, TVN_GETDISPINFO,

TVN_GETINFOTIP, TVN_ITEMEXPANDED, TVN_ITEMEXPANDING, TVN_KEYDOWN, TVN_SELCHANGED,

TVN_SELCHANGING, TVN_SETDISPINFO, and TVN_SINGLEEXPAND.

Check your Visual C++ online documentation for more information about handling these notification messages. Now is a good time to again build and test Common. Be sure to try expanding and collapsing the levels of the tree and editing a label. If you can't see all the control, maximize the application and adjust your screen resolution if you can. The application will eventually scroll but not just yet.

The Rich Edit Control

If you took all the energy expended on writing text-editing software and you concentrated that energy on other, less mundane programming problems, computer science would probably be a decade ahead of where it is now. Although that might be an exaggeration, it is true that when it comes to text editors, a huge amount of effort has been dedicated to reinventing the wheel.

Wouldn't it be great to have one piece of text-editing code that all programmers could use as the starting point for their own custom text editors? With Visual C++'s CRichEditCtrl control, you get a huge jump on any text-editing functionality that you need to install in your applications. The rich edit control is capable of handling fonts, paragraph styles, text color, and other types of tasks that are traditionally found in text editors. In fact, a rich edit control (named for the fact that it handles text in Rich Text Format) provides a solid starting point for any text-editing tasks that your application must handle. Your users can

- Type text.
- Edit text, using cut-and-paste and sophisticated drag-and-drop operations.
- Set text attributes such as font, point size, and color.
- Apply underline, bold, italic, strikethrough, superscript, and subscript properties to text.
- Format text, using various alignments and bulleted lists.
- Lock text from further editing.
- Save and load files.

As you can see, a rich edit control is powerful. It is, in fact, almost a complete word-processorin- a-box that you can plug into your program and use immediately. Of course, because a rich edit control offers so many features, there's a lot to learn. This section gives you a quick introduction to creating and manipulating a rich edit control.

Creating the Rich Edit Control

Add a call to CreateRichEdit() to the view class's OnCreate() function and then add the function to the class. Listing 10.10 shows the code you should add to the function. Add resource

IDs for IDC_RICHEDIT, IDC_RICHEDIT_ULINE, IDC_RICHEDIT_LEFT, IDC_RICHEDIT_CENTER, and IDC_RICHEDIT_RIGHT.

Listing 10.10 CommonView.cpp—CCommonView::CreateRichEdit()

```
void CCommonView::CreateRichEdit()
{
        m_richEdit.Create(WS_CHILD | WS_VISIBLE | WS_BORDER |
        ES_AUTOVSCROLL | ES_MULTILINE,
        CRect(180, 260, 393, 360), this, IDC_RICHEDIT);
        m_boldButton.Create("ULine", WS_VISIBLE | WS_CHILD | WS_BORDER,
        CRect(400, 260, 450, 280), this, IDC_RICHEDIT_ULINE);
        m_leftButton.Create("Left", WS_VISIBLE | WS_CHILD | WS_BORDER,
        CRect(400, 285, 450, 305), this, IDC_RICHEDIT_LEFT);
        m_centerButton.Create("Center", WS_VISIBLE | WS_CHILD | WS_BORDER,
        CRect(400, 310, 450, 330), this, IDC_RICHEDIT_CENTER);
        m_rightButton.Create("Right", WS_VISIBLE | WS_CHILD | WS_BORDER,
        CRect(400, 335, 450, 355), this, IDC_RICHEDIT_RIGHT);
}
```

As usual, things start with a call to the control's Create() member function. The style constants include the same constants that you would use for creating any type of window, with the addition of special styles used with rich edit controls. Table 10.5 lists these special styles.

Table 10.5 Rich Edit Styles

Style	Description
ES_AUTOHSCROLL	Automatically scrolls horizontally
ES_AUTOVSCROLL	Automatically scrolls vertically
ES_CENTER	Centers text
ES_LEFT	Left-aligns text
ES_LOWERCASE	Lowercases all text
ES_MULTILINE	Enables multiple lines
ES_NOHIDESEL	Doesn't hide selected text when losing the focus
ES_OEMCONVERT	Converts from ANSI characters to OEM characters and back to ANSI
ES_PASSWORD	Displays characters as asterisks
ES_READONLY	Disables editing in the control

ES_RIGHT	Right-aligns text
ES_UPPERCASE	Uppercases all text
ES_WANTRETURN	Inserts return characters into text when Enter is pressed

Initializing the Rich Edit Control

The rich edit control is perfectly usable as soon as it is created. Member functions manipulate the control extensively, formatting and selecting text, enabling and disabling many control features, and more. As always, check your online documentation for all the details on these member functions.

Manipulating the Rich Edit Control

This sample application shows you the basics of using the rich edit control by setting character attributes and paragraph formats. When you include a rich edit control in an application, you will probably want to give the user some control over its contents. For this reason, you usually create menu and toolbar commands for selecting the various options that you want to support in the application. In Common, the user can click four buttons to control the rich edit control. You've already added the code to create these buttons. Add lines to the message map in the header file to declare the handlers:

```
afx_msg void OnULine();
afx_msg void OnLeft();
afx_msg void OnCenter();
afx_msg void OnRight();
```

Similarly, add these lines to the message map in the source file:

```
ON_COMMAND(IDC_RICHEDIT_ULINE, OnULine)
ON_COMMAND(IDC_RICHEDIT_LEFT, OnLeft)
ON_COMMAND(IDC_RICHEDIT_CENTER, OnCenter)
ON_COMMAND(IDC_RICHEDIT_RIGHT, OnRight)
```

Each of these functions is simple. Add them each to CommonView.cpp. OnULine() looks

like this:

```
void CCommonView::OnULine()
{
    CHARFORMAT charFormat;
    charFormat.cbSize = sizeof(CHARFORMAT);
    charFormat.dwMask = CFM_UNDERLINE;
```

```
    m_richEdit.GetSelectionCharFormat(charFormat);
    if (charFormat.dwEffects & CFM_UNDERLINE)
        charFormat.dwEffects = 0;
    else
        charFormat.dwEffects = CFE_UNDERLINE;
        m_richEdit.SetSelectionCharFormat(charFormat);
        m_richEdit.SetFocus();
}
```

OnULine() creates and initializes a CHARFORMAT structure, which holds information about character formatting and is declared in Listing 10.11.

Listing 10.11 The CHARFORMAT Structure, Defined by MFC

```
typedef struct _charformat
{
    UINT cbSize;
    _WPAD _wPad1;
    DWORD dwMask;
    DWORD dwEffects;
        LONG yHeight;
    LONG yOffset;
    COLORREF crTextColor;
    BYTE bCharSet;
    BYTE bPitchAndFamily;
    TCHAR szFaceName[LF_FACESIZE];
    _WPAD _wPad2;
} CHARFORMAT;
```

In a CHARFORMAT structure, cbSize is the size of the structure. dwMask indicates which members of the structure are valid (can be a combination of CFM_BOLD, CFM_CHARSET, CFM_COLOR, CFM_FACE, CFM_ITALIC, CFM_OFFSET, CFM_PROTECTED, CFM_SIZE, CFM_STRIKEOUT, and CFM_UNDERLINE). dwEffects is the character effects (can be a combination of CFE_AUTOCOLOR, CFE_BOLD, CFE_ITALIC, CFE_STRIKEOUT, CFE_UNDERLINE, and CFE_PROTECTED). yHeight is the character height, and yOffset is the character baseline offset (for super- and subscript characters).

crTextColor is the text color. bCharSet is the character set value (see the ifCharSet member of the LOGFONT structure). bPitchAndFamily is the font pitch and family, and

szFaceName is the font name.

After initializing the CHARFORMAT structure, as needed, to toggle underlining, OnULine() calls the control's GetSelectionCharFormat() member function. This function, whose single argument is a reference to the CHARFORMAT structure, fills the character format structure. OnULine() checks the dwEffects member of the structure to determine whether to turn underlining on or off. The bitwise and operator, &, is used to test a single bit of the variable.Finally, after setting the character format, OnULine() returns the focus to the rich edit control. By clicking a button, the user has removed the focus from the rich edit control. You don't want to force the user to keep switching back manually to the control after every button click, so you do it by calling the control's SetFocus() member function. Common also enables the user to switch between the three types of paragraph alignment. This is accomplished similarly to toggling character formats. Listing 10.12 shows the three functions— OnLeft(), OnRight(), and OnCenter()—that handle the alignment commands. Add the code for these functions to CommonView.cpp. As you can see, the main difference is the use of the PARAFORMAT structure instead of CHARFORMAT and the call to SetParaFormat() instead of SetSelectionCharFormat().

Listing 10.12 CommonView.cpp—Changing Paragraph Formats

```
void CCommonView::OnLeft()
{
    PARAFORMAT paraFormat;
    paraFormat.cbSize = sizeof(PARAFORMAT);
    paraFormat.dwMask = PFM_ALIGNMENT;
    paraFormat.wAlignment = PFA_LEFT;
    m_richEdit.SetParaFormat(paraFormat);
    m_richEdit.SetFocus();
}
void CCommonView::OnCenter()
{
    PARAFORMAT paraFormat;
    paraFormat.cbSize = sizeof(PARAFORMAT);
    paraFormat.dwMask = PFM_ALIGNMENT;
    paraFormat.wAlignment = PFA_CENTER;
    m_richEdit.SetParaFormat(paraFormat);
    m_richEdit.SetFocus();
```

```
}
void CCommonView::OnRight()
{
        PARAFORMAT paraFormat;
        paraFormat.cbSize = sizeof(PARAFORMAT);
        paraFormat.dwMask = PFM_ALIGNMENT;
        paraFormat.wAlignment = PFA_RIGHT;
        m_richEdit.SetParaFormat(paraFormat);
        m_richEdit.SetFocus();
}
```

After adding all that code, it's time to build and test again. First, click in the text box to give it the focus. Then, start typing. Want to try out character attributes? Click the ULine button to add underlining to either selected text or the next text you type. To try out paragraph formatting, click the Left, Center, or Right button to specify paragraph alignment. (Again, if you're using large text, adjust the button size if the labels don't fit.) Figure 10.9 shows the rich edit control with some different character and paragraph styles used.

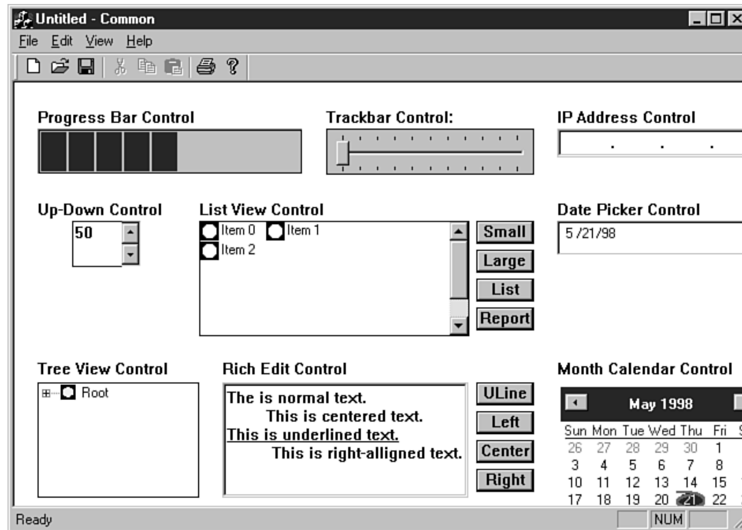


FIG. 10.9 A rich edit control is almost a complete word processor.

IP Address Control

If you're writing an Internet-aware program, you might have already wondered how you're going to validate certain kinds of input from your users. One thing you could ask for is an IP address, like this one: 205.210.40.1

IP addresses always have four parts, separated by dots, and each part is always a number between 1 and 255. The IP address picker guarantees that the user will give

you information that meets this format. To try it out, add yet another line to OnCreate(), this time a call to CreateIPAddress(). Add the function to the class. The code is really simple; just add a call to Create():

```
void CCommonView::CreateIPAddress()
{
    m_ipaddress.Create(WS_CHILD | WS_VISIBLE | WS_BORDER,
    CRect(470,40,650,65), this, IDC_IPADDRESS);
}
```

Remember to add a resource ID for IDC_IPADDRESS. No special styles are related to this simple control. There are some useful member functions to get, set, clear, or otherwise manipulate the address. Check them out in the online documentation.

Build and run Common, and try entering numbers or letters into the parts of the field. Notice how the control quietly fixes bad values (enter 999 into one part, for example) and how it moves you along from part to part as you enter the third digit or type a dot. It's a simple control, but if you need to obtain IP addresses from the user, this is the only way to fly.

The Date Picker Control

How many different applications ask users for dates? It can be annoying to have to type a date according to some preset format. Many users prefer to click on a calendar to select a day. Others find this very slow and would rather type the date, especially if they're merely changing an existing date. The date picker control, in the MFC class CDateTimeCtrl, gives your users the best of both worlds. Start, as usual, by adding a call to CreateDatePicker() to CCommonView::OnCreate() and then adding the function to the class. Add the resource ID for IDC_DATE. Like the IP Address control, the date picker needs only to be created. Add this code to CommonView.cpp:

```
void CCommonView::CreateDatePicker()
{
    m_date.Create(WS_CHILD | WS_VISIBLE | DTS_SHORTDATEFORMAT,
    CRect(470,120,650,150), this, IDC_DATE);
}
```

The CDateTimeCtrl class, of which m_date is an object, defines special styles to be used with date picker controls. Table 10.6 lists these special styles.

Table 10.6 Date Picker Control Styles

Style	Description
DTS_APPCANPARSE	Instructs the date control to give more control to your application while the user edits dates.

DTS_LONGDATEFORMAT	After the date is picked, displays it like Monday, May 18, 1998 or whatever your locale has defined for long dates.
DTS_RIGHTALIGN	Aligns the calendar with the right edge of the control (if you don'tspecify this style, it will align with the left edge).
DTS_SHOWNONE	A date is optional: A check box indicates that a date has been selected.
DTS_SHORTDATEFORMAT	After the date is picked, displays it like 5/18/98 or whatever your locale has defined for short dates.
DTS_TIMEFORMAT date.	Displays the time as well as the
DTS_UPDOWN	Uses an up-down control instead of a calendar for picking.

There are a number of member functions that you might use to set colors and fonts for this control, but the most important function is GetTime(), which gets you the date and time entered by the user. It fills in a COleDateTime or CTime object, or a SYSTEMTIME structure, which you can access by individual members. Here's the declaration of SYSTEMTIME:

```
typedef struct _SYSTEMTIME {
        WORD wYear;
        WORD wMonth;
        WORD wDayOfWeek;
        WORD wDay;
        WORD wHour;
        WORD wMinute;
        WORD wSecond;
        WORD wMilliseconds;
} SYSTEMTIME;
```

If you want to do anything with this date, you're probably going to find it easier to work with as a CTime object. The CTime class is discussed in Appendix F, "Useful Classes." For now, you probably just want to see how easy it is to use the control, so build and test Common yet again. Click the drop-down box next to the short date, and you will see how the date picker got its name. Choose a date and see the short date change. Edit the date and then drop the month down again, and you will see that the highlight has moved to the day you entered. Notice, also, that today's date is circled on

the month part of this control. This month calendar is a control of its own. One is created by the date picker, but you will create another one in the next section.

Month Calendar Control

The month calendar control used by the date picker is compact and neat. Putting one into Common is very simple. Add a call to CreateMonth() to CCommonView::OnCreate() and add the function to the class. Add a resource ID for IDC_MONTH, too; then add the code for CreateMonth(). Here it is:

```
void CCommonView::CreateMonth()
{
    m_month.Create(WS_CHILD | WS_VISIBLE |
DTS_SHORTDATEFORMAT,
    CRect(470,260,650,420), this, IDC_MONTH);
}
```

You can use many of the DTS_ styles when creating your month calendar control. In addition, the CMonthCalCtrl class, of which m_month is an object, defines special styles to be used with month calendar controls. Table 10.7 lists these special styles.

Table 10.7 Month Calendar Control Styles

Style	Description
MCS_DAYSTATE	Instructs the control to send MCN_GETDAYSTATE messages to the application so that special days (such as holidays) can be displayed in bold.
MCS_MULTISELECT	Enables the user to choose a range of dates.
MCS_NOTODAY	Suppresses the Today date at the bottom of the control. The user can display today's date by clicking the word *Today*.
MCS_NOTODAY_CIRCLE	Suppresses the circling of today's date.
MCS_WEEKNUMBERS	Numbers each week in the year from 1 to 52 and displays the numbers at the left of the calendar.

A number of member functions enable you to customize the control, setting the colors, fonts, and whether weeks start on Sunday or Monday. You will be most interested in GetCurSel(), which fills a COleDateTime, CTime, or LPSYSTEMTIME with the currently selected date. Build and test Common again and really exercise the month control this time. (Make the window larger if you can't see the whole control.) Try moving from month to month. If you're a long way from today's date, click the Today down at the bottom to return quickly. This is a neat control and should quickly replace the various third-party calendars that so many developers have been using.

Scrolling the View

After adding all these controls, you might find that they don't all fit in the window. As Figure 10.10 shows, no scrollbars appear, even though CCommonView inherits from CScrollView. You need to set the scroll sizes in order for scrolling to work properly.

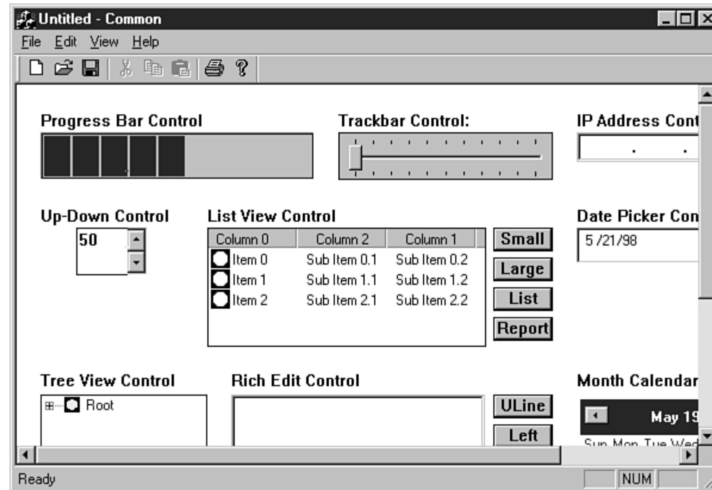


FIG. 10.10 The view doesn't automatically gain scrollbars as more controls are added.

Expand CCommonView and double-click OnInitialUpdate() in ClassView. Edit it so that it looks like this:

```
void CCommonView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    CSize sizeTotal;
    sizeTotal.cx = 700;
    sizeTotal.cy = 500;
    SetScrollSizes(MM_TEXT, sizeTotal);
}
```

The last control you added, the month calendar, ran from the coordinates (470, 260) to (650, 420). This code states that the entire document is 700´500 pixels, so it leaves a nice white margin between that last control and the edge of the view. When the displayed window is less than 700´500, you get scrollbars. When it's larger, you don't. The call to SetScrollSizes() takes care of all the work involved in making scrollbars, sizing them to represent the proportion of the document that is displayed, and dealing with the user's scrollbar clicks. Try it yourself— build Common one more time and experiment with resizing it and scrolling around. (The scrollbars weren't there before because the OnInitialUpdate() generated by AppWizard stated that the app was 100´100 pixels, which wouldn't require scrollbars.)

So, what's going on? Vertical scrolling is fine, but horizontal scrolling blows up your application, right? You can use the techniques described in Appendix D, "Debugging," to find the cause. The problem is in OnHScroll(), which assumed that any horizontal scrolling was related to the slider control and acted accordingly. Edit that function so that it looks like this:

```
void CCommonView::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    CSliderCtrl* slider = (CSliderCtrl*)pScrollBar;
    if (slider == &m_trackbar)
    {
        int position = slider->GetPos();
        char s[10];
        wsprintf(s, "%d ", position);
        CClientDC clientDC(this);
        clientDC.TextOut(390, 22, s);
    }
    CScrollView::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

Now the slider code is executed only when the scrollbar that was clicked is the one kept in m_trackbar. The rest of the time, the work is simply delegated to the base class. For the last time, build and test Common—everything should be perfect now.

UNIT – V

ACTIVEX CONCEPTS

In this chapter

The Purpose of ActiveX

Object Linking

Object Embedding

Containers and Servers

Toward a More Intuitive User Interface

The Component Object Model

Automation

ActiveX Controls

The Purpose of ActiveX

This chapter covers the theory and concepts of ActiveX, which is built on the Component Object Model (COM). Until recently, the technology built on COM was called OLE, and OLE still exists, but the emphasis now is on ActiveX. Most new programmers have found OLE intimidating, and the switch to ActiveX is unlikely to lessen that. However, if you think of ActiveX technology as a way to use code already written and tested by someone else, and as a way to save yourself the trouble of reinventing the wheel, you'll see why it's worth learning. Developer Studio and MFC make ActiveX much easier to understand and implement by doing much of the groundwork for you. There are four chapters in Part V, "Internet Programming," and together they demonstrate what ActiveX has become. In addition, ActiveX controls, which to many developers represent the way of the future, are discussed in Chapter 20, "Building an Internet ActiveX Control," and Chapter 21, "The Active Template Library."

Windows has always been an operating system that allows several applications running at once, and right from the beginning, programmers wanted to have a way for those applications to exchange information while running. The Clipboard was a marvelous innovation, though, of course, the user had to do a lot of the work. DDE (Dynamic Data Exchange) allowed applications to "talk" to each other but had some major limitations. Then came OLE 1 (Object Linking and Embedding). Later there was OLE 2, and then Microsoft just called it OLE, until it moved so far beyond its original roots that it was renamed ActiveX.

ActiveX lets users and applications be document-centered, and this is probably the most important thing about it. If a user wants to create an annual report, by choosing ActiveX-enabled applications, the user stays focused on that annual report.

Perhaps parts of it are being done with Word and parts with Excel, but, to the user, these applications are not really the point. This shift in focus is happening on many fronts and corresponds to a more object-oriented way of thinking among many programmers. It seems more natural now to share work among several different applications and arrange for them to communicate than to write one huge application that can do everything.

Here's a simple test to see whether you are document centered or application centered: How is your hard drive organized?

The directory structure in Figure 13.1 is application centered: The directories are named for the applications that were used to create the documents they hold. All Word documents are together, even though they might be for very different clients or projects.

Microsoft Office
Word
Building Internet Apps
Using Visual C++
Acme Corp
Training
Web Pages
Excel
Journal
Sales estimates
Invoices
ABC Inc
Payroll System
Inventory System
Microsoft Developer Studio
ABC Inc Payroll System
ABC Inc Inventory System

FIG. 13.1 An application-centered directory structure arranges documents by type.

The directory structure in Figure 13.2 is document centered: The directories are named for the client or project involved. All the sales files are together, even though they can be accessed with a variety of different applications.

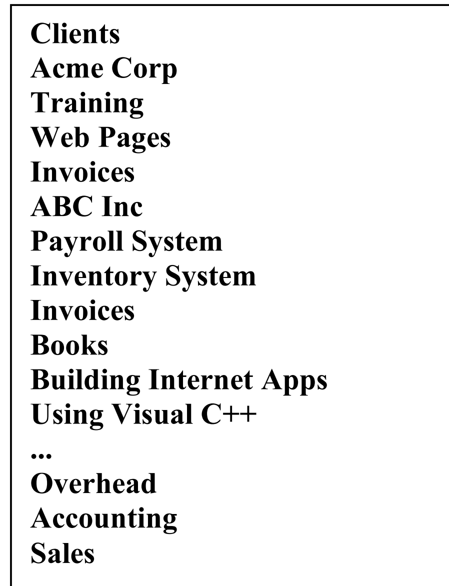


FIG. 13.2 A document-centered directory structure arranges documents by meaning or content.

If you've been using desktop computers long enough, you remember when using a program involved a program disk and a data disk. Perhaps you remember installing software that demanded to know the data directory where you would keep all the files created with that product. That was application-centered thinking, and it's fast being supplanted by documentcentered thinking.

Why? What's wrong with application-centered thinking? Well, where do you put the documents that are used with two applications equally often? There was a time when each product could read its own file formats and no others. But these days, the lines between applications are blurring; a document created in one word processor can easily be read into another, a spreadsheet file can be used as a database, and so on. If a client sends you a WordPerfect document and you don't have WordPerfect, do you make a \WORDPERFECT\DOCS directory to put it in, or add it to your \MSOFFICE\WORD\DOCS directory? If you have your hard drive arranged in a more document-centered manner, you can just put it in the directory for that client.

The Windows 95 interface, now incorporated into Windows NT as well, encourages document centered thinking by having users double-click documents to automatically launch the applications that created them. This wasn't new—File Manager had that capability for years—but it feels very different to double-click an icon that's just sitting on the desktop than it does to start an application and then double-click an entry in a list box. More and more it doesn't matter what application or applications were involved in creating this document; you just want to see and change your data, and you want to do that quickly and simply.

After you become document-centered, you see the appeal of *compound documents*—files created with more than one application. If your report needs an illustration, you create it in some graphic program and then stick it in with your text when it's done. If your annual report needs a table, and you already have the numbers in a spreadsheet, you don't retype them into the table feature of your word processor or even import them; you incorporate them as a spreadsheet excerpt, right in the middle of your text. This isn't earth-shatteringly new, of course. Early desktop publishing programs such as Ventura pulled together text and graphics from a variety of sources into one complex compound document. What's exciting is being able to do it simply, intuitively, and with so many different applications.

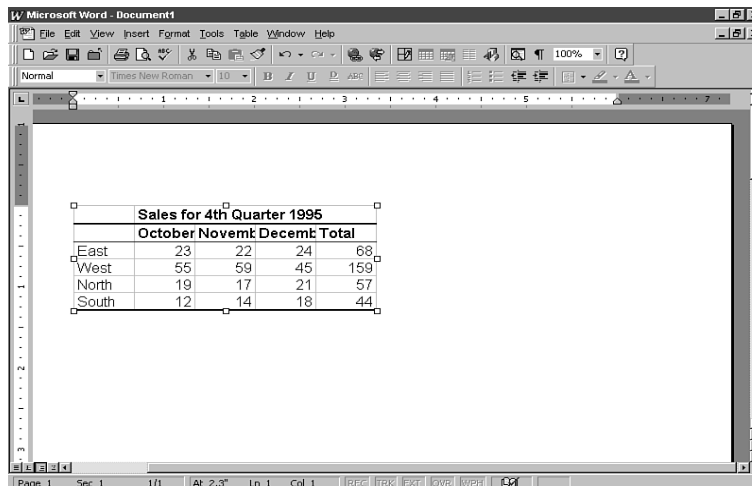
Object Linking

Figure 13.3 shows a Word document with an Excel spreadsheet linked into it.

Follow these steps to create a similar document yourself:

1. Start Word and enter your text.
2. Click where you want the table to go.
3. Choose Insert, Object.
4. Select the Create from File tab.
5. Enter or select the filename as though this were a File Open dialog box.
6. Be sure to check the Link to File box.
7. Click OK.

The entire file appears in your document. If you make a change in the file on disk, the change is reflected in your document. You can edit the file in its own application by double-clicking it within Word. The other application is launched to edit it, as shown in Figure 13.4. If you delete the file from disk, your Word document still displays what the file last looked like, but you aren't able to edit it.



	Sales for 4th Quarter 1995			
	October	Novemb	Decemb	Total
East	23	22	24	68
West	55	59	45	159
North	19	17	21	57
South	12	14	18	44

FIG. 13.3 A Microsoft Word document can contain a link to an Excel file.

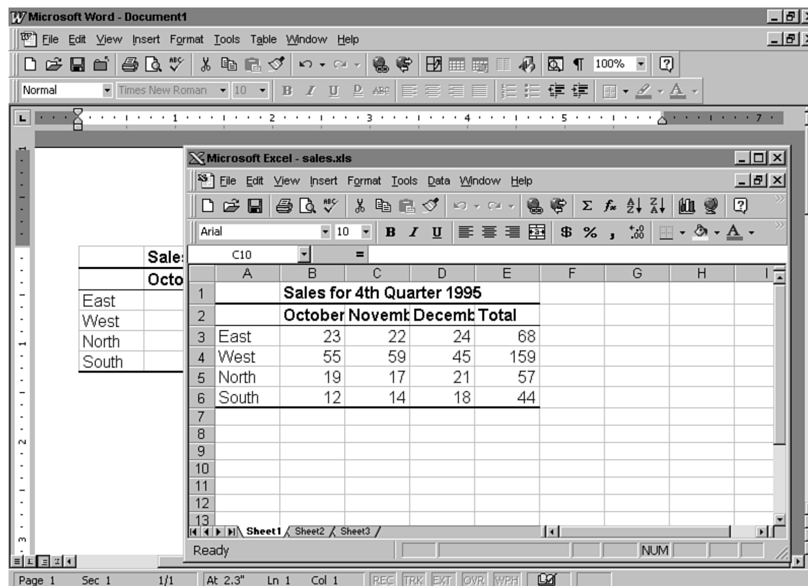


FIG. 13.4 Double-clicking a linked object launches the application that created it.

You link files into your documents if you plan to use the same file in many documents and contexts, because your changes to that file are automatically reflected everywhere that you have linked it. Linking doesn't increase the size of your document files dramatically because only the location of the file and a little bit of presentation information needs to be kept in your document.

Object Embedding

Embedding is similar to linking, but a copy of the object is made and placed into your document. If you change the original, the changes aren't reflected in your document. You can't tell by looking whether the Excel chart you see in your Word document is linked or embedded. Figure 13.5 shows a spreadsheet embedded within a Word document.

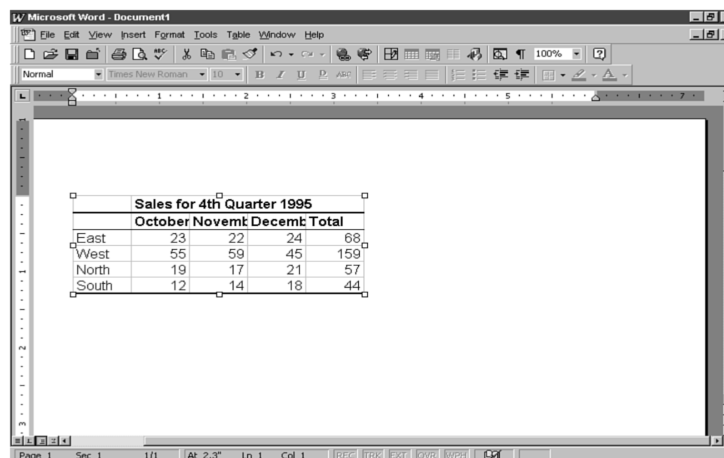


FIG. 13.5 A file embedded within another file looks just like a linked file.

Follow these steps to create a similar document yourself:

1. Start Word and enter your text.
2. Click where you want the table to go.
3. Choose Insert, Object.
4. Select the Create from File tab.
5. Enter or select the filename as though this were a File Open dialog box.
6. Do not check the Link to File box.
7. Click OK.

What's the difference? You'll see when you double-click the object to edit it. The Word menus and toolbars disappear and are replaced with their Excel equivalents, as shown in Figure 13.6. Changes you make here aren't made in the file you originally embedded. They are made in the copy of that file that has become part of your Word document.

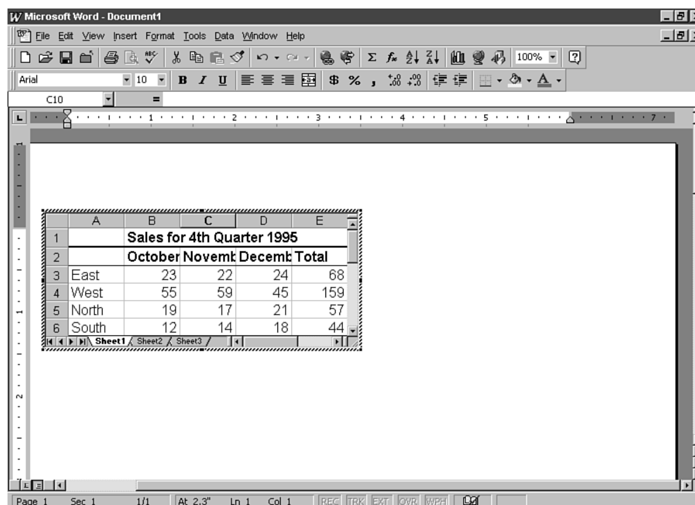


FIG. 13.6 Editing in place is the magic of OLE embedding.

You embed files into your documents if you plan to build a compound document and then use it as a self-contained whole, without using the individual parts again. Changes you make don't affect any other files on your disk, not even the one you copied from in the first place. Embedding makes your document much larger than it was, but you can delete the original if space is a problem.

Containers and Servers

To embed or link one object into another, you need a *container* and a *server*. The container is the application into which the object is linked or embedded—Word in these examples. The server is the application that made them, and that can be launched (perhaps in place) when the object is double-clicked—Excel in these examples.

Why would you develop a container application? To save yourself work. Imagine you have a product already developed and in the hands of your users. It does a specific task like organize a sales team, schedule games in a league sport, or calculate life insurance rates. Then your users tell you that they wish it had a spreadsheet capability so they could do small calculations on the fly. How long will it take you to add that functionality? Do you really have time to learn how spreadsheet programs parse the functions that users type? If your application is a container app, it doesn't take any time at all. Tell your users to link or embed in an Excel sheet and let Excel do the work. If they don't own a copy of Excel, they need some spreadsheet application that can be an ActiveX server. You get to piggyback on the effort of other developers.

It's not just spreadsheets, either. What if users want a scratch pad, a place to scribble a few notes? Let them embed a Word document. (What about bitmaps and other illustrations? Microsoft Paint, or a more powerful graphics package if they have one, and it can act as an ActiveX server.) You don't have to concern yourself with adding functionality like this to your programs because you can just make your application a container and your users can embed whatever they want without any more work on your part. Why would you develop a server application, then? Look back over the reasons for writing a container application. A lot of users are going to contact developers asking for a feature to be added, and be told they can have that feature immediately—they just need an application that does spreadsheets, text, pictures, or whatever, and can act as an ActiveX server. If your application is an ActiveX server, people will buy it so that they can add its functionality to their container apps.

Together, container and server apps enable users to build the documents they want. They represent a move toward building-block software and a document-centered approach to work. If you want your application to carry the Windows 95 logo, it must be a server, a container, or both. But there is much more to ActiveX than linking and embedding.

Toward a More Intuitive User Interface

What if the object you want to embed is not in a file but is part of a document you have open at the moment? You may have already discovered that you can use the Clipboard to transfer ActiveX objects. For example, to embed part of an Excel spreadsheet into a Word document, you can follow these steps:

1. Open the spreadsheet in Excel.
2. Open the document in Word.
3. In Excel, select the portion you want to copy.
4. Choose Edit, Copy to copy the block onto the Clipboard.
5. Switch to Word and choose Edit, Paste Special.
6. Select the Paste radio button.
7. Select Microsoft Excel Worksheet Object from the list box.
8. Make sure that Display as Icon is not selected.
9. The dialog box should look like Figure 13.7. Click OK.

A copy of the block is now embedded into the document. If you choose Paste Link, changes in the spreadsheet are reflected immediately in the Word document, not

just when you save them. (You might have to click the selection in Word to update it.) This is true even if the spreadsheet has no name and has never been saved. Try it yourself! This is certainly better than saving dummy files just to embed them into compound documents and then deleting them, isn't it?

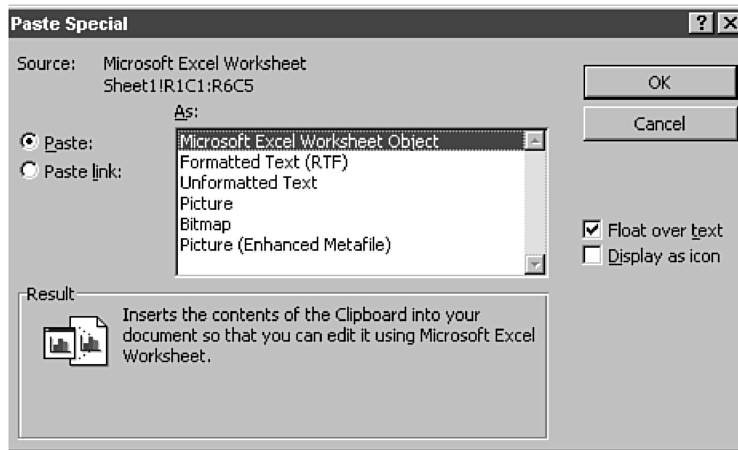


FIG. 13.7 The Paste Special dialog box is used to link or embed selected portions of a document.

Another way to embed part of a document into another is drag and drop. This is a user interface paradigm that works in a variety of contexts. You click something (an icon, a highlighted block of text, a selection in a list box) and hold the mouse button down while moving it. The item you clicked moves with the mouse, and when you let go of the mouse button, it drops to the new location. That's very intuitive for moving or resizing windows, but now you can use it to do much, much more. For example, here's how that Excel-in-Word example would be done with drag and drop:

1. Open Word and size it to less than full screen.

2. Open Excel and size it to less than full screen. If you can arrange the Word and Excel windows so they don't overlap, that's great.

3. In Excel, select the portion you want to copy by highlighting it with the mouse or cursor keys.

4. Click the border of the selected area (the thick black line) and hold.

5. Drag the block into the Word window and let go.

The selected block is embedded into the Word document. If you double-click it, you are editing in place with Excel. Drag and drop also works within a document to move or copy a selection.You can also use drag and drop with icons. On your desktop, if you drag a file to a folder, it is moved there. (Hold down Ctrl while dragging to copy it.) If you drag it to a program icon, it is opened with that program. This is very useful when you have a document you use with two applications. For example, pages on the World Wide Web are HTML documents, often created with an HTML editor but viewed with a World Wide Web browser such as Netscape Navigator or Microsoft Internet Explorer.

If you double-click an HTML document icon, your browser is launched to view it. If you drag that icon onto the icon for your HTML editor, the editor is launched and

opens the file you dragged. After you realize you can do this, you will find your work speeds up dramatically. All of this is ActiveX, and all of this requires a little bit of work from programmers to make it happen. So what's going on?

The Component Object Model

The heart of ActiveX is the Component Object Model (COM). This is an incredibly complex topic that deserves a book of its own. Luckily, the Microsoft Foundation Classes and the Visual C++ AppWizard do much of the behind-the-scenes work for you. The discussion in these chapters is just what you need to know to use COM as a developer. COM is a binary standard for Windows objects. That means that the executable code (in a DLL or EXE) that describes an object can be executed by other objects. Even if two objects were written in different languages, they are able to interact using the COM standard.

How do they interact? Through an *interface*. An ActiveX interface is a collection of functions, or really just function names. It's a C++ class with no data, only pure virtual functions. Your objects inherit from this class and provide code for the functions. (Remember, as discussed in Appendix A, "C++ Review and Object-Oriented Concepts," a class that inherits a pure virtual function doesn't inherit code for that function.) Other programs get to your code by calling these functions. All ActiveX objects must have an interface named IUnknown (and most have many more, all with names that start with I, the prefix for interfaces).

The IUnknown interface has only one purpose: finding other interfaces. It has a function called QueryInterface() that takes an interface ID and returns a pointer to that interface for thisobject. All the other interfaces inherit from IUnknown, so they have a QueryInterface() too, and you have to write the code—or you would if there was no MFC. MFC implements a number of macros that simplify the job of writing interfaces and their functions, as you will shortly see. The full declaration of IUnknown is in Listing 13.1. The macros take care of some of the work of declaring an interface and won't be discussed here. There are three functions declared:

QueryInterface(), AddRef(), and Release().

These latter two functions are used to keep track of which applications are using an interface. All three functions are inherited by all interfaces and must be implemented by the developer of the interface.

Listing 13.1 IUnknown, Defined in \Program Files\Microsoft Visual

Studio\VC98\Include\unknwn.h

```
MIDL_INTERFACE("00000000-0000-0000-C000-000000000046")
IUnknown
{
    public:
    BEGIN_INTERFACE
        virtual HRESULT STDMETHODCALLTYPE QueryInterface(
    /* [in] */ REFIID riid,
    /* [iid_is][out] */ void __RPC_FAR *__RPC_FAR *ppvObject) = 0;
```

```
        virtual ULONG STDMETHODCALLTYPE AddRef( void) = 0;
        virtual ULONG STDMETHODCALLTYPE Release( void) = 0;
        #if (_MSC_VER >= 1200) // VC6 or greater
        template <class Q>
        HRESULT STDMETHODCALLTYPE QueryInterface(Q** pp)
        {
                return QueryInterface(__uuidof(Q), (void**)pp);
        }
        #endif
        END_INTERFACE
};
```

Automation

An Automation server lets other applications tell it what to do. It *exposes* functions and data, called *methods* and *properties*. For example, Microsoft Excel is an Automation server, and programs written in Visual C++ or Visual Basic can call Excel functions and set properties like column widths. That means you don't need to write a scripting language for your application any more. If you expose all the functions and properties of your application, any programming language that can control an Automation server can be a scripting language for your application. Your users may already know your scripting language. They essentially will have no learning curve for writing macros to automate your application (although they will need to learn the names of the methods and properties you expose).

The important thing to know about interacting with automation is that one program is always in control, calling the methods or changing the properties of the other running application. The application in control is called an Automation controller. The application that exposes methods and functions is called an Automation server. Excel, Word, and other members of the Microsoft Office suite are Automation servers, and your programs can use the functions of these applications to really save you coding time.

For example, imagine being able to use the function called by the Word menu item Format, Change Case to convert the blocks of text your application uses to all uppercase, all lowercase, sentence case (the first letter of the first word in each sentence is uppercase, the rest are not), or title case (the first letter of every word is uppercase; the rest are not).

The description of how automation really works is far longer and more complex than the interface summary of the previous section. It involves a special interface called IDispatch, a simplified interface that works from a number of different languages, including those like Visual Basic that can't use pointers. The declaration of IDispatch is shown in Listing 13.2.

Listing 13.2 IDispatch, Defined in \Program Files\Microsoft Visual Studio\VC98\Include\oaidl.h

```
MIDL_INTERFACE("00020400-0000-0000-C000-000000000046")
IDispatch : public IUnknown
{
    public:
        virtual HRESULT STDMETHODCALLTYPE GetTypeInfoCount(
            /* [out] */ UINT __RPC_FAR *pctinfo) = 0;

        virtual HRESULT STDMETHODCALLTYPE GetTypeInfo(
            /* [in] */ UINT iTInfo,
            /* [in] */ LCID lcid,
        /* [out] */ ITypeInfo __RPC_FAR *__RPC_FAR *ppTInfo) = 0;
    virtual HRESULT STDMETHODCALLTYPE GetIDsOfNames(
        /* [in] */ REFIID riid,
        /* [size_is][in] */ LPOLESTR __RPC_FAR *rgszNames,
        /* [in] */ UINT cNames,
        /* [in] */ LCID lcid,
        /* [size_is][out] */ DISPID __RPC_FAR *rgDispId) = 0;
    virtual /* [local] */ HRESULT STDMETHODCALLTYPE Invoke(
        /* [in] */ DISPID dispIdMember,
        /* [in] */ REFIID riid,
        /* [in] */ LCID lcid,
        /* [in] */ WORD wFlags,
        /* [out][in] */ DISPPARAMS __RPC_FAR *pDispParams,
        /* [out] */ VARIANT __RPC_FAR *pVarResult,
        /* [out] */ EXCEPINFO __RPC_FAR *pExcepInfo,
        /* [out] */ UINT __RPC_FAR *puArgErr) = 0;
};
```

Although IDispatch seems more complex than IUnknown, it declares only a few more functions:

GetTypeInfoCount(), GetTypeInfo(), GetIDsOfNames(), and Invoke().

Because it inherits from IUnknown, it has also inherited QueryInterface(), AddRef(), and Release(). They are all pure virtual functions, so any COM class that inherits from

IDispatch must implement these functions. The most important of these is Invoke(), used to call functions of the Automation server and to access its properties.

ActiveX Controls

ActiveX controls are tiny little Automation servers that load *in process*. This means they are remarkably fast. They were originally called OLE Custom Controls and were designed to replace VBX controls, 16-bit controls written for use in Visual Basic and Visual C++. (There are a number of good technical reasons why the VBX technology could not be extended to the 32-bit world.) Because OLE Custom Controls were traditionally kept in files with the extension .OCX, many people referred to an OLE Custom Control as an OCX control or just an OCX. Although the OLE has been supplanted by ActiveX, ActiveX controls produced by Visual C++ 6.0 are still kept in files with the .OCX extension. The original purpose of VBX controls was to allow programmers to provide unusual interface controls to their users. Controls that looked like gas gauges or volume knobs became easy to develop. But almost immediately, VBX programmers moved beyond simple controls to modules that involved significant amounts of calculation and processing. In the same way, many ActiveX controls are far more than just controls; they are *components* that can be used to build powerful applications quickly and easily.

Because controls are little Automation servers, they need to be used by an Automation controller, but the terminology is too confusing if there are controls and controllers, so we say that ActiveX controls are used by *container* applications. Visual C++ and Visual Basic are both container applications, as are many members of the Office suite and many non-Microsoft products. In addition to properties and methods, ActiveX controls have *events*. To be specific, a control is said to *fire* an event, and it does so when there is something that the container needs to be aware of. For example, when the user clicks a portion of the control, the control deals with it, perhaps changing its appearance or making a calculation, but it may also need to pass on word of that click to the container application so that a file can be opened or some other container action can be performed.

This chapter has given you a brief tour through the concepts and terminology used in ActiveX technology, and a glimpse of the power you can add to your applications by incorporating ActiveX into them. The remainder of the chapters in this part lead you through the creation of ActiveX applications, using MFC and the wizards in Visual C++.

BUILDING AN ACTIVEX CONTAINER APPLICATION

In this chapter

Changing ShowString

Moving, Resizing, and Tracking

Handling Multiple Objects and Object Selection

Implementing Drag and Drop

Deleting an Object

You can obtain a rudimentary ActiveX container by asking AppWizard to make you one, but it will have a lot of shortcomings. A far more difficult task is to understand how an ActiveX container works and what you have to do to really use it. In this chapter, by turning the ShowString application of earlier chapters into an ActiveX container and then making it a truly functional container, you get a backstage view of ActiveX in action. Adding drag-and-drop support brings your application into the modern age of intuitive, document-centered user interface design. If you have not yet read Chapter 13, “ActiveX Concepts,” it would be a good idea to read it before this one. As well, this chapter will not repeat all the instructions of Chapter 8, “Building a Complete Application: ShowString,” so you should have read that chapter or be prepared to refer to it as you progress through this one.

Changing ShowString

ShowString was built originally in Chapter 8, “Building a Complete Application: ShowString,” and has no ActiveX support. You could make the changes by hand to implement ActiveX container support, but there would be more than 30 changes. It’s quicker to build a new ShowString application—this time asking for ActiveX container support—and then make changes to that code to get the ShowString functionality again.

AppWizard-Generated ActiveX Container Code

Build the new ShowString in a different directory, making almost exactly the same AppWizard choices you used when you built it in the “Creating an Empty Shell with AppWizard” section of Chapter 8. Name the project ShowString, choose an MDI Application, No Database Support, compound document support: Container, a Docking Toolbar, Initial Status Bar, Printing and Print Preview, Context Sensitive Help, and 3D Controls. Finally, select Source File Comments and a Shared DLL. Finish AppWizard and, if you want, build the project.tm1713714470

There are many differences between the application you just built and a do-nothing application without ActiveX container support. The remainder of this section explains these differences and their effects.

Menus There's another menu, called IDR_SHOWSTTYPE_CNTR_IP, shown in Figure 14.1. The name refers to a container whose *contained* object is being edited *in place*. During in-place editing, the menu bar is built from the container's in-place menu and the server's in-place menu. The pair of vertical bars in the middle of IDR_SHOWSTTYPE_CNTR_IP are separators; the server menu items will be put between them. This is discussed in more detail in Chapter 15, "Building an ActiveX Server Application."

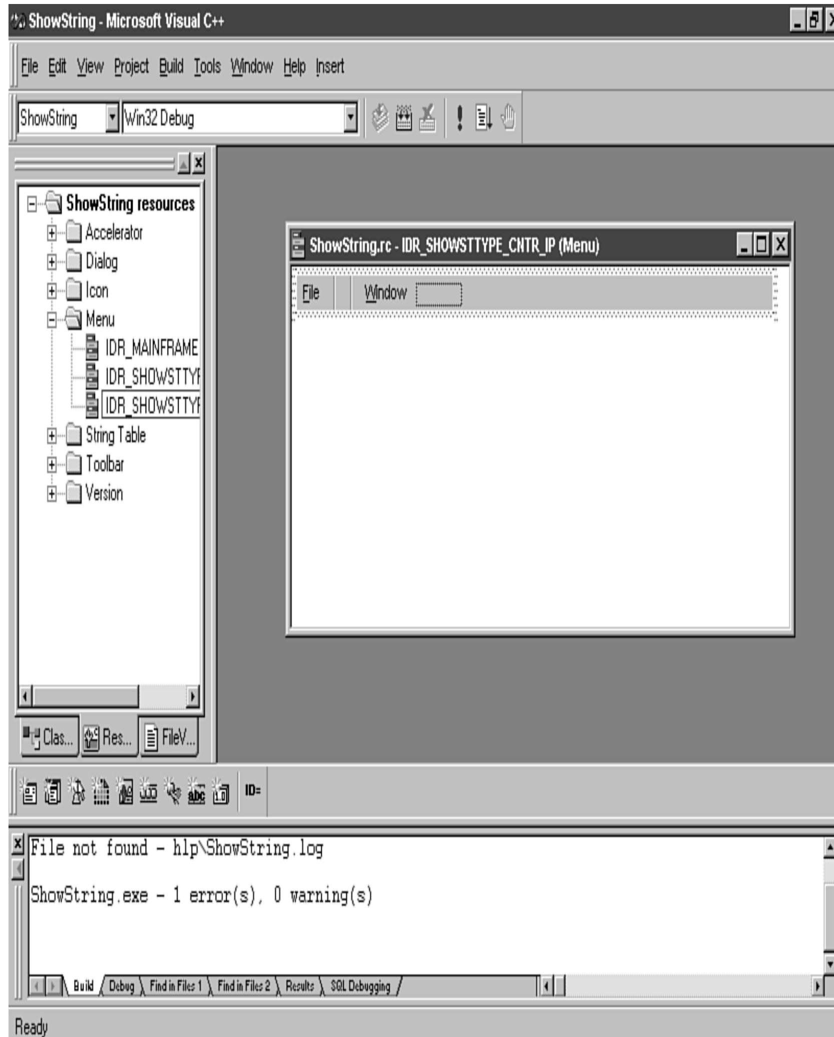


FIG. 14.1 AppWizard adds another menu for editing in place.

The IDR_SHOWSTTYPE Edit menu, shown in Figure 14.2, has four new items:

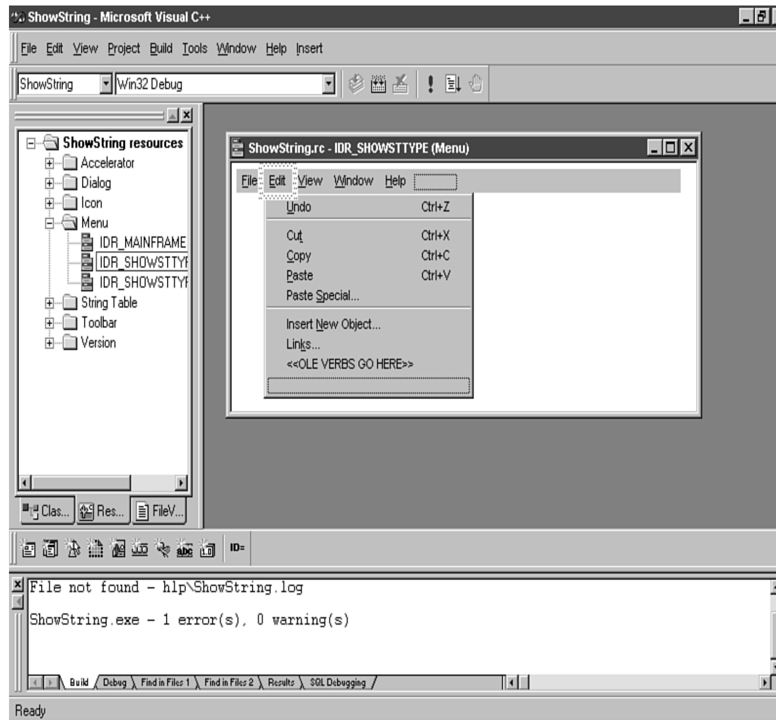


FIG. 14.2 AppWizard adds items to the Edit menu of the IDR_SHOWSTTYPE resource.

- **Paste Special.** The user chooses this item to insert an item into the container from the Clipboard.
- **Insert New Object.** Choosing this item opens the Insert Object dialog box, shown in Figures 14.3 and 14.4, so the user can insert an item into the container.

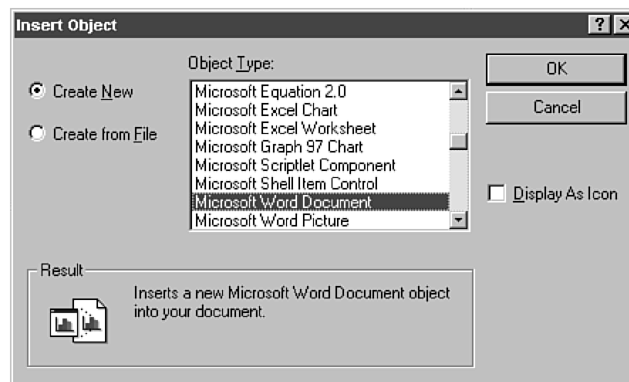


FIG. 14.3 The Insert Object dialog box can be used to embed new objects.

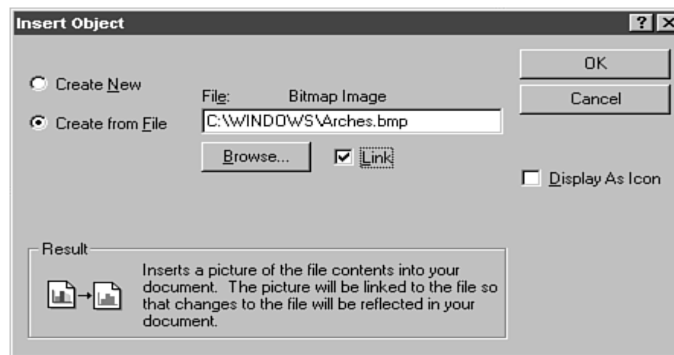


FIG. 14.4 The Insert Object dialog box can be used to embed or link objects that are in a file.

- **Links.** When an object has been linked into the container, choosing this item opens the Links dialog box, shown in Figure 14.5, to allow control of how the copy of the object is updated after a change is saved to the file.
- **<<OLE VERBS GO HERE>>.** Each kind of item has different verbs associated with it, like Edit, Open, or Play. When a contained item has focus, this spot on the menu is replaced by an object type like those in the Insert Object dialog box, with a menu cascading from it that lists the verbs for this type, like the one shown in Figure 14.6.

CShowStringApp CShowStringApp::InitInstance() has several changes from the InitInstance() method provided by AppWizard for applications that aren't ActiveX containers.The lines in Listing 14.1 initialize the ActiveX (OLE) libraries.

Listing 14.1 Excerpt from ShowString.cpp—Library Initialization

```
// Initialize OLE libraries
        if (!AfxOleInit())
        {
                AfxMessageBox(IDP_OLE_INIT_FAILED);
                return FALSE;
        }
```

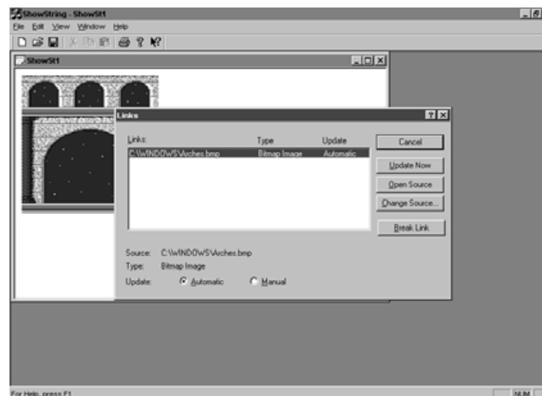


FIG. 14.5 The Links dialog box controls the way linked objects are updated.

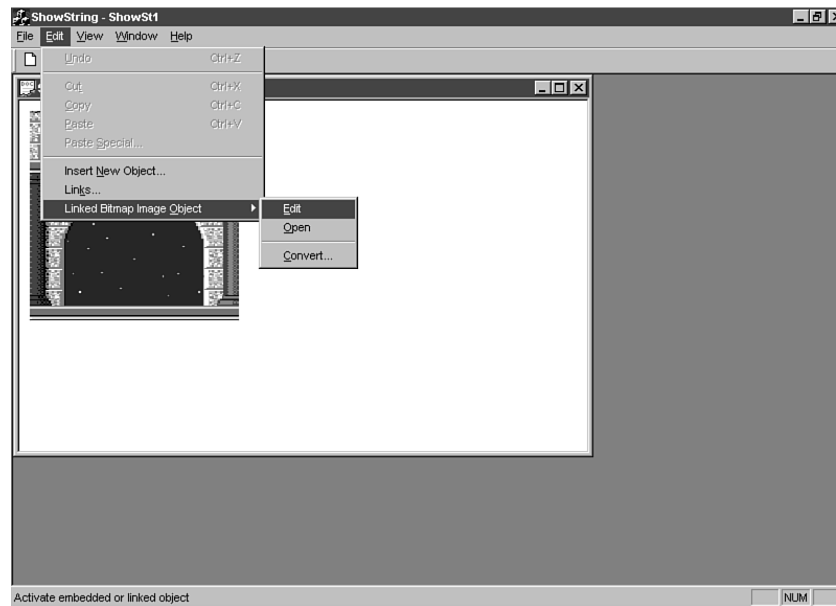


FIG. 14.6 Each object type adds a cascading menu item to the Edit menu when it has focus.

Still in CShowStringApp::InitInstance(), after the MultiDocTemplate is initialized but before the call to AddDocTemplate(), this line is added to register the menu used for in-place editing:

```
pDocTemplate->SetContainerInfo(IDR_SHOWSTTYPE_CNTR_IP);
```

CShowStringDoc The document class, CShowStringDoc, now inherits from COleDocument rather than CDocument. This line is also added at the top of ShowStringDoc.cpp:

```
#include "CntrItem.h"
```

CntrItem.h describes the container item class, CShowStringCntrItem, discussed later in this chapter. Still in ShowStringDoc.cpp, the macros in Listing 14.2 have been added to the message map.

Listing 14.2 Excerpt from ShowString.cpp—Message Map Additions

```
ON_UPDATE_COMMAND_UI(ID_EDIT_PASTE,
        COleDocument::OnUpdatePasteMenu)
ON_UPDATE_COMMAND_UI(ID_EDIT_PASTE_LINK,
        COleDocument::OnUpdatePasteLinkMenu)
ON_UPDATE_COMMAND_UI(ID_OLE_EDIT_CONVERT,
        COleDocument::OnUpdateObjectVerbMenu)
ON_COMMAND(ID_OLE_EDIT_CONVERT,
```

```
    COleDocument::OnEditConvert)
ON_UPDATE_COMMAND_UI(ID_OLE_EDIT_LINKS,
    COleDocument::OnUpdateEditLinksMenu)
ON_COMMAND(ID_OLE_EDIT_LINKS,
    COleDocument::OnEditLinks)
ON_UPDATE_COMMAND_UI(ID_OLE_VERB_FIRST, ID_OLE_VERB_LAST,
    COleDocument::OnUpdateObjectVerbMenu)
```

These commands enable and disable the following menu items:

- Edit, Paste
- Edit, Paste Link
- Edit, Links
- The OLE verbs section, including the Convert verb

The new macros also handle Convert and Edit, Links. Notice that the messages are handled by functions of COleDocument and don't have to be written by you. The constructor, CShowStringDoc::CShowStringDoc(), has a line added:

```
EnableCompoundFile();
```

This turns on the use of compound files. CShowStringDoc::Serialize() has a line added as well:

```
COleDocument::Serialize(ar);
```

This call to the base class Serialize() takes care of serializing all the contained objects, with no further work for you.

CShowStringView The view class, CShowStringView, includes CntrItem.h just as the document does. The view class has these new entries in the message map:

```
ON_WM_SETFOCUS()
ON_WM_SIZE()
ON_COMMAND(ID_OLE_INSERT_NEW, OnInsertObject)
ON_COMMAND(ID_CANCEL_EDIT_CNTR, OnCancelEditCntr)
```

These are in addition to the messages caught by the view before it was a container. These catch WM_SETFOCUS, WM_SIZE, the menu item Edit, Insert New Object, and the cancellation of editing in place. An accelerator has already been added to connect this message to the Esc key. In ShowStringView.h, a new member variable has been added, as shown in Listing 14.3.

Listing 14.3 Excerpt from ShowStringView.h—m_pSelection

```
// m_pSelection holds the selection to the current
// CShowStringCntrItem. For many applications, such
```

```
// a member variable isn't adequate to represent a
// selection, such as a multiple selection or a selection
// of objects that are not CShowStringCntrItem objects.
// This selection mechanism is provided just to help you
// get started.
// TODO: replace this selection mechanism with one appropriate
// to your app.
CShowStringCntrItem* m_pSelection;
```

This new member variable shows up again in the view constructor, Listing 14.4, and the revised OnDraw(), Listing 14.5.

Listing 14.4 ShowStringView.cpp—Constructor

```
CShowStringView::CShowStringView()
{
    m_pSelection = NULL;
    // TODO: add construction code here
}
```

Listing 14.5 ShowStringView.cpp—CShowStringView::OnDraw()

```
void CShowStringView::OnDraw(CDC* pDC)
{
    CShowStringDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    // TODO: also draw all OLE items in the document
    // Draw the selection at an arbitrary position. This code should be
    // removed once your real drawing code is implemented. This position
    // corresponds exactly to the rectangle returned by
    CShowStringCntrItem,
    // to give the effect of in-place editing.
    // TODO: remove this code when final draw code is complete.
    if (m_pSelection == NULL)
    {
        POSITION pos = pDoc->GetStartPosition();
```

```
            m_pSelection = (CShowStringCntrItem*)pDoc-
            >GetNextClientItem(pos);
        }

        if (m_pSelection != NULL)
        m_pSelection->Draw(pDC, CRect(10, 10, 210, 210));
}
```

The code supplied for OnDraw() draws only a single contained item. It doesn't draw any native data—in other words, elements of ShowString that are not contained items. At the moment there is no native data, but after the string is added to the application, OnDraw() is going to have to draw it. What's more, this code only draws one contained item, and it does so in an arbitrary rectangle. OnDraw() is going to see a lot of changes as you work through this chapter.

The view class has gained a lot of new functions. They are as follows:

- OnInitialUpdate()
- IsSelected()
- OnInsertObject()
- OnSetFocus()
- OnSize()
- OnCancelEditCntr()

Each of these new functions is discussed in the subsections that follow.

OnInitialUpdate() OnInitialUpdate()is called just before the very first time the view is to be displayed. The boilerplate code (see Listing 14.6) is pretty dull.

Listing 14.6 ShowStringView.cpp—CShowStringView::OnInitialUpdate()

```
void CShowStringView::OnInitialUpdate()
{
        CView::OnInitialUpdate();
        // TODO: remove this code when final selection
        // model code is written
        m_pSelection = NULL; // initialize selection
}
```

The base class OnInitialUpdate() calls the base class OnUpdate(), which calls Invalidate(), requiring a full repaint of the client area.

IsSelected() IsSelected() currently isn't working because the selection mechanism is so rudimentary. Listing 14.7 shows the code that was generated for you. Later, when you have implemented a proper selection method, you will improve how this code works.

Listing 14.7 ShowStringView.cpp—CShowStringView::IsSelected()

```
BOOL CShowStringView::IsSelected(const CObject* pDocItem) const
{
    // The implementation below is adequate if your selection consists of
    // only CShowStringCntrItem objects. To handle different selection
    // mechanisms, the implementation here should be replaced.
    // TODO: implement this function that tests for a selected OLE
    // client item
    return pDocItem == m_pSelection;
}
```

This function is passed a pointer to a container item. If that pointer is the same as the current selection, it returns TRUE.

OnInsertObject() OnInsertObject()is called when the user chooses Edit, Insert New Object. It's quite a long function, so it is presented in parts. The overall structure is presented in Listing 14.8.

Listing 14.8 ShowStringView.cpp—CShowStringView::OnInsertObject()

```
    void CShowStringView::OnInsertObject()
    {
        // Display the Insert Object dialog box.
        CShowStringCntrItem* pItem = NULL;
        TRY
        {
            // Create a new item connected to this document.
            // Initialize the item.
            // Set selection and update all views.
        }
        CATCH(CException, e)
        {
            // Handle failed create.
        }
        END_CATCH
        // Tidy up.
    }
```

First, this function displays the Insert Object dialog box, as shown in Listing 14.9.

Listing 14.9 ShowStringView.cpp—Display the Insert Object Dialog Box

```
// Invoke the standard Insert Object dialog box to obtain information
// for new CShowStringCntrItem object.
COleInsertDialog dlg;
if (dlg.DoModal() != IDOK)
return;
BeginWaitCursor();
```

If the user clicks Cancel, this function returns and nothing is inserted. If the user clicks OK, the cursor is set to an hourglass while the rest of the processing occurs. To create a new item, the code in Listing 14.10 is inserted.

Listing 14.10 ShowStringView.cpp—Create a New Item

```
// Create new item connected to this document.
CShowStringDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
pItem = new CShowStringCntrItem(pDoc);
ASSERT_VALID(pItem);
```

This code makes sure there is a document, even though the menu item is enabled only if there is one, and then creates a new container item, passing it the pointer to the document. As you see in the CShowStringCntrItem section, container items hold a pointer to the document that contains them.

The code in Listing 14.11 initializes that item.

Listing 14.11 ShowStringView.cpp—Initializing the Inserted Item

```
// Initialize the item from the dialog data.
    if (!dlg.CreateItem(pItem))
        AfxThrowMemoryException(); // any exception will do
        ASSERT_VALID(pItem);
// If item created from class list (not from file) then launch
// the server to edit the item.
if (dlg.GetSelectionType() == COleInsertDialog::createNewItem)
    pItem->DoVerb(OLEIVERB_SHOW, this);

    ASSERT_VALID(pItem);
```

The code in Listing 14.11 calls the CreateItem() function of the dialog class, COleInsertDialog. That might seem like a strange place to keep such a function, but the function needs to know all the answers that were given on the dialog box. If it was a member of another class, it would have to interrogate the dialog for the type and filename, find out whether it was linked or embedded, and so on. It calls member functions of the container item like CreateLinkFromFile(), CreateFromFile(), CreateNewItem(), and so on. So it's not that the code has to actually fill the object from the file that is in the dialog box, but rather that the work is partitioned between the objects instead of passing information back and forth between them.

Then, one question is asked of the dialog box: Was this a new item? If so, the server is called to edit it. Objects created from a file can just be displayed.

Finally, the selection is updated and so are the views, as shown in Listing 14.12.

Listing 14.12 ShowStringView.cpp—Update Selection and Views

```
// As an arbitrary user interface design, this sets the selection
// to the last item inserted.
// TODO: reimplement selection as appropriate for your application
m_pSelection = pItem; // set selection to last inserted item
pDoc->UpdateAllViews(NULL);
```

If the creation of the object failed, execution ends up in the CATCH block, shown in Listing 14.13.

Listing 14.13 ShowStringView.cpp—CATCH Block

```
CATCH(CException, e)
{
    if (pItem != NULL)
    {
        ASSERT_VALID(pItem);
        pItem->Delete();
    }
    AfxMessageBox(IDP_FAILED_TO_CREATE);
}
END_CATCH
```

This deletes the item that was created and gives the user a message box. Finally, that hourglass cursor can go away:

```
EndWaitCursor();
```

OnSetFocus() OnSetFocus(), shown in Listing 14.14, is called whenever this view gets focus.

Listing 14.14 ShowStringView.cpp—CShowStringView::OnSetFocus()

```
void CShowStringView::OnSetFocus(CWnd* pOldWnd)
{
    COleClientItem* pActiveItem = GetDocument()-> GetInPlaceActiveItem(this);
    if (pActiveItem != NULL &&
    pActiveItem->GetItemState() == COleClientItem::activeUIState)
    {
        // need to set focus to this item if it is in the same view
        CWnd* pWnd = pActiveItem->GetInPlaceWindow();
        if (pWnd != NULL)
        {
            pWnd->SetFocus(); // don't call the base class
            return;
        }
    }
    CView::OnSetFocus(pOldWnd);
}
```

If there is an active item and its server is loaded, that active item gets focus. If not, focus remains with the old window, and it appears to the user that the click was ignored.

OnSize() OnSize(), shown in Listing 14.15, is called when the application is resized by the user.

Listing 14.15 ShowStringView.cpp—CShowStringView::OnSize()

```
void CShowStringView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    COleClientItem* pActiveItem = GetDocument()-> GetInPlaceActiveItem(this);
    if (pActiveItem != NULL)
        pActiveItem->SetItemRects();
}
```

This resizes the view using the base class function, and then, if there is an active item, tells it to adjust to the resized view.

OnCancelEditCntr() OnCancelEditCntr() is called when a user who has been editing in place presses Esc. The server must be closed, and the object stops being active. The code is shown in Listing 14.16.

Listing 14.16 ShowStringView.cpp—CShowStringView::OnCancelEditCntr()

```
void CShowStringView::OnCancelEditCntr()
{
    // Close any in-place active item on this view.
    COleClientItem* pActiveItem =
    GetDocument()->GetInPlaceActiveItem(this);
    if (pActiveItem != NULL)
    {
        pActiveItem->Close();
    }
    ASSERT(GetDocument()->GetInPlaceActiveItem(this) == NULL);
}
```

CShowStringCntrItem The container item class is a completely new addition to ShowString. It describes an item that is contained in the document. As you've already seen, the document and the view use this object quite a lot, primarily through the m_pSelection member variable of CShowStringView. It has no member variables other than those inherited from the base class, COleClientItem. It has overrides for a lot of functions, though. They are as follows:

- A constructor
- A destructor
- GetDocument()
- GetActiveView()
- OnChange()
- OnActivate()
- OnGetItemPosition()
- OnDeactivateUI()
- OnChangeItemPosition()
- AssertValid()
- Dump()
- Serialize()

The constructor simply passes the document pointer along to the base class. The destructor does nothing. GetDocument() and GetActiveView() are inline functions that return member variables inherited from the base class by calling the base class function with the same name and casting the result. OnChange() is the first of these functions that has more than one line of code (see Listing 14.17).

Listing 14.17 CntrItem.cpp—CShowStringCntrItem::OnChange()

```
void CShowStringCntrItem::OnChange(OLE_NOTIFICATION nCode,
DWORD dwParam)
{
    ASSERT_VALID(this);
    COleClientItem::OnChange(nCode, dwParam);
    // When an item is being edited (either in-place or fully open)
    // it sends OnChange notifications for changes in the state of the
    // item or visual appearance of its content.
    // TODO: invalidate the item by calling UpdateAllViews
    // (with hints appropriate to your application)
    GetDocument()->UpdateAllViews(NULL);
    // for now just update ALL views/no hints
    }
```

Actually, there are only three lines of code. The comments are actually more useful than the code. When the user changes the contained item, the server notifies the container. Calling UpdateAllViews() is a rather drastic way of refreshing the screen, but it gets the job done. OnActivate() (shown in Listing 14.18) is called when a user double-clicks an item to activate it and edit it in place. ActiveX objects are usually *outside-in*, which means that a single click of the item selects it but doesn't activate it. Activating an outside-in object requires a double-click, or a single click followed by choosing the appropriate OLE verb from the Edit menu.

Listing 14.18 CntrItem.cpp—CShowStringCntrItem::OnActivate()

```
void CShowStringCntrItem::OnActivate()
{
    // Allow only one in-place activate item per frame
    CShowStringView* pView = GetActiveView();
    ASSERT_VALID(pView);
    COleClientItem* pItem = GetDocument()->
    GetInPlaceActiveItem(pView);
```

```
	if (pItem != NULL && pItem != this)
	pItem->Close();
	COleClientItem::OnActivate();
}
```

This code makes sure that the current view is valid, closes the active items, if any, and then activates this item. OnGetItemPosition() (shown in Listing 14.19) is called as part of the in-place activation process.

Listing 14.19 CntrItem.cpp—CShowStringCntrItem::OnGetItemPosition()

```
void CShowStringCntrItem::OnGetItemPosition(CRect& rPosition)
{
	ASSERT_VALID(this);
	// During in-place activation,
	// CShowStringCntrItem::OnGetItemPosition
	// will be called to determine the location of this item.
	// The default implementation created from AppWizard simply
	// returns a hard-coded rectangle. Usually, this rectangle
	// would reflect the current position of the item relative
	// to the view used for activation. You can obtain the view
	// by calling CShowStringCntrItem::GetActiveView.
	// TODO: return correct rectangle (in pixels) in rPosition
	rPosition.SetRect(10, 10, 210, 210);
}
```

Like OnChange(), the comments are more useful than the actual code. At the moment, the View’s OnDraw() function draws the contained object in a hard-coded rectangle, so this function returns that same rectangle. You are instructed to write code that asks the active view where the object is. OnDeactivateUI() (see Listing 14.20) is called when the object goes from being active to inactive.

Listing 14.20 CntrItem.cpp—CShowStringCntrItem::OnDeactivateUI()

```
void CShowStringCntrItem::OnDeactivateUI(BOOL bUndoable)
{
	COleClientItem::OnDeactivateUI(bUndoable);
	// Hide the object if it is not an outside-in object
	DWORD dwMisc = 0;
```

```
        m_lpObject->GetMiscStatus(GetDrawAspect(), &dwMisc);
        if (dwMisc & OLEMISC_INSIDEOUT)
                DoVerb(OLEIVERB_HIDE, NULL);
}
```

Although the default behavior for contained objects is outside-in, as discussed earlier, you can write *inside-out objects*. These are activated simply by moving the mouse pointer over them; clicking the object has the same effect that clicking that region has while editing the object. For example, if the contained item is a spreadsheet, clicking might select the cell that was clicked. This can be really nice for the user, who can completely ignore the borders between the container and the contained item, but it is harder to write. OnChangeItemPosition() is called when the item is moved during in-place editing. It, too, contains mostly comments, as shown in Listing 14.21.

Listing 14.21 CntrItem.cpp—CShowStringCntrItem::OnChangeItemPosition()

```
BOOL CShowStringCntrItem::OnChangeItemPosition(const CRect& rectPos)
{
        ASSERT_VALID(this);
        // During in-place activation
        // CShowStringCntrItem::OnChangeItemPosition
        // is called by the server to change the position
        // of the in-place window. Usually, this is a result
        // of the data in the server document changing such that
        // the extent has changed or as a result of in-place resizing.
        //
        // The default here is to call the base class, which will call
        // COleClientItem::SetItemRects to move the item
        // to the new position.
        if (!COleClientItem::OnChangeItemPosition(rectPos))
                return FALSE;
        // TODO: update any cache you may have of the item's rectangle/extent

        return TRUE;
}
```

This code is supposed to handle moving the object, but it doesn't really. That's because OnDraw() always draws the contained item in the same place. AssertValid() is

a debug function that confirms this object is valid; if it's not, an ASSERT will fail. ASSERT statements are discussed in Chapter 24, "Improving Your Application's Performance." The last function in CShowStringCntrItem is Serialize(), which is called by COleDocument::Serialize(), which in turn is called by the document's Serialize(), as you've already seen. It is shown in Listing 14.22.

Listing 14.22 CntrItem.cpp—CShowStringCntrItem::Serialize()

```
void CShowStringCntrItem::Serialize(CArchive& ar)
{
        ASSERT_VALID(this);
        // Call base class first to read in COleClientItem data.
        // Because this sets up the m_pDocument pointer returned from
        // CShowStringCntrItem::GetDocument, it is a good idea to call
        // the base class Serialize first.
        COleClientItem::Serialize(ar);
        // now store/retrieve data specific to CShowStringCntrItem
        if (ar.IsStoring())
        {
                // TODO: add storing code here
        }
        else
        {
                // TODO: add loading code here
        }
}
```

All this code does at the moment is call the base class function. COleDocument::Serialize() stores or loads a number of counters and numbers to keep track of several different contained items, and then calls helper functions such as WriteItem() or ReadItem() to actually deal with the item. These functions and the helper functions they call are a bit too "behind-the-scenes" for most people, but if you'd like to take a look at them, they are in the MFC source folder (C:\Program Files\Microsoft Visual Studio\VC98\MFC\SRC on many installations) in the file olecli1.cpp. They do their job, which is to serialize the contained item for you.

Shortcomings of This Container This container application isn't ShowString yet, of course, but it has more important things wrong with it. It isn't a very good container, and that's a direct result of all those TODO tasks that haven't been accomplished. Still, the fact that it is a functioning container is a good measure of the power of the MFC classes COleDocument and COleClientItem. So why not build the application

now and run it? After it's running, choose Edit, Insert New Object and insert a bitmap image. Now that you've seen the code, it shouldn't be a surprise that Paint is immediately launched to edit the item in place, as you see in Figure 14.7.

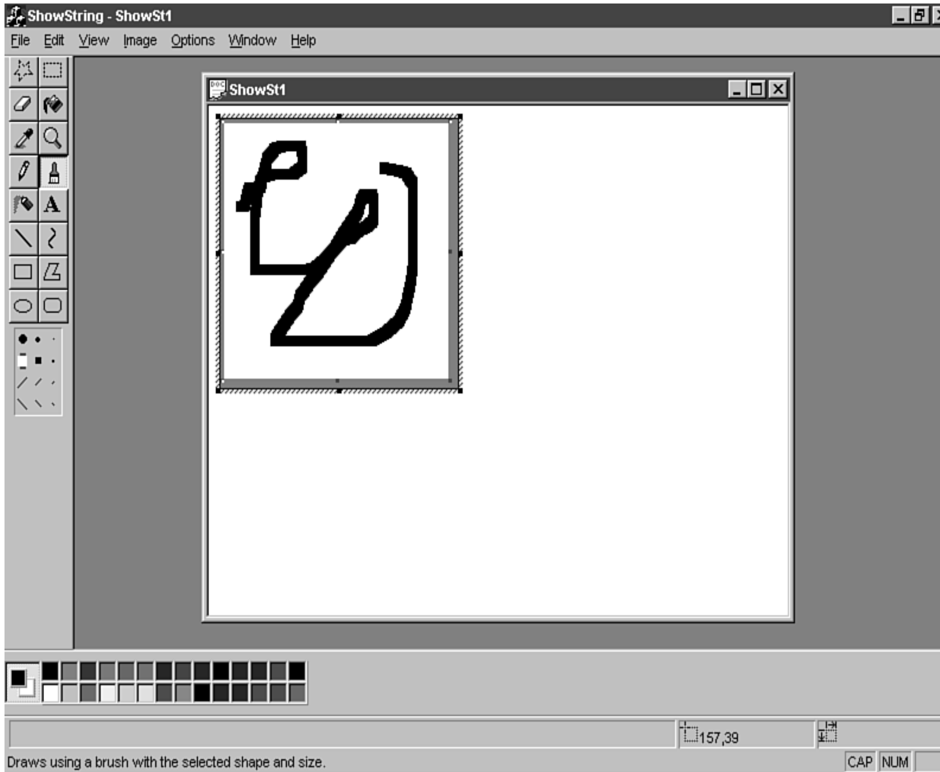


FIG. 14.7 The boilerplate container can contain items and activate them for in-place editing, like this bitmap image being edited in Paint.

Click outside the bitmap to deselect the item and return control to the container; you see that nothing happens. Click outside the document, and again nothing happens. You're probably asking yourself, "Am I still in ShowString?" Choose File, New, and you see that you are. The Paint menus and toolbars go away, and a new ShowString document is created. Click the bitmap item again, and you are still editing it in Paint. How can you insert another object into the first document when the menus are those of Paint? Press Esc to cancel in-place editing so the menus become ShowString menus again. Insert an Excel chart into the container, and the bitmap disappears as the new Excel chart is inserted, as shown in Figure 14.8. Obviously, this container leaves a lot to be desired.

Press Esc to cancel the in-place editing, and notice that the view changes a little, as shown in Figure 14.9. That's because CShowStringView::OnDraw() draws the contained item in a 200´200 pixel rectangle, so the chart has to be squeezed a little to fit into that space. It is the server—Excel, in this case—that decides how to fit the item into the space given to it by the container.

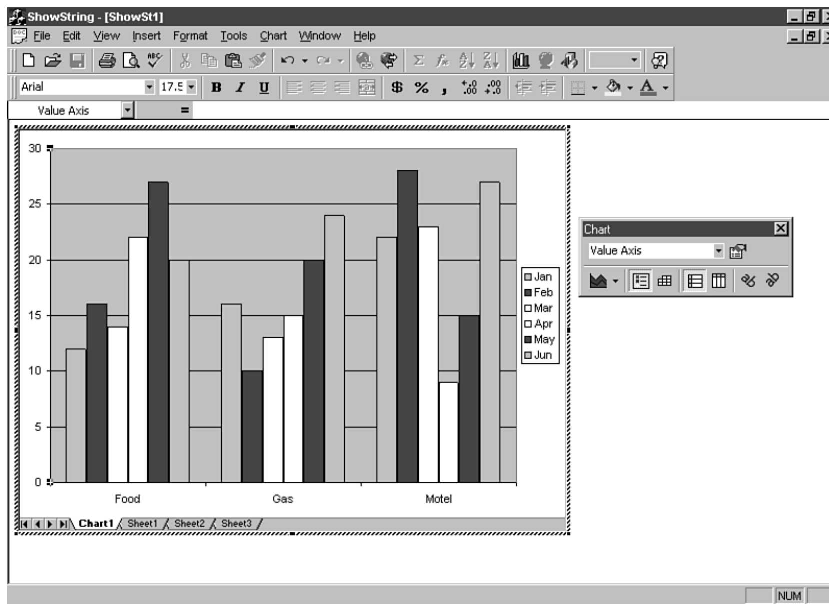


FIG. 14.8 Inserting an Excel chart gets you a default chart, but it completely covers the old bitmap.

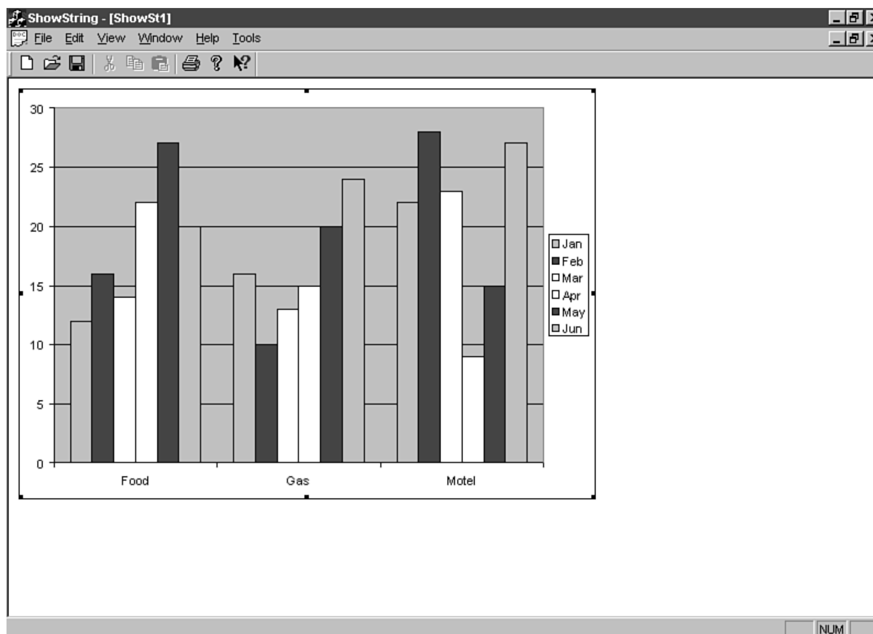


FIG. 14.9 Items can look quite different when they are not active.

As you can see, there's a lot to be done to make this feel like a real container. But first, you have to turn it back into ShowString.

Returning the ShowString Functionality

This section provides a quick summary of the steps presented in Chapter 8, "Building a Complete Application: ShowString." Open the files from the old ShowString as you go so that you can copy code and resources wherever possible. Follow these steps:

1. In ShowStringDoc.h, add the private member variables and public Get functions to the class.
2. In CShowStringDoc::Serialize(), paste the code that saves or restores these member variables. Leave the call to COleDocument::Serialize() in place.
3. In CShowStringDoc::OnNewDocument(), paste the code that initializes the member variables.
4. In CShowStringView::OnDraw(), add the code that draws the string before the code that handles the contained items. Remove the TODO task about drawing native data.
5. Copy the Tools menu from the old ShowString to the new container ShowString. Choose File, Open to open the old ShowString.rc, open the IDR_SHOWSTTYPE menu, click the Tools menu, and choose Edit, Copy. Open the new ShowString's IDR_SHOWSTTYPE menu, click the Window menu, and choose Edit, Paste. Don't paste it into the IDR_SHOWSTTYPE_CNTR_IP menu.
6. Add the accelerator Ctrl+T for ID_TOOLS_OPTIONS as described in Chapter 8, "Building a Complete Application: ShowString." Add it to the IDR_MAINFRAME accelerator only.
7. Delete the IDD_ABOUTBOX dialog box from the new ShowString. Copy IDD_ABOUTBOX and IDD_OPTIONS from the old ShowString to the new.
8. While IDD_OPTIONS has focus, choose View, Class Wizard. Create the COptionsDialog class as in the original ShowString.
9. Use the Class Wizard to connect the dialog controls to COptionsDialog member variables, as described in Chapter 10.
10. Use the Class Wizard to arrange for CShowStringDoc to catch the ID_TOOLS_OPTIONS command.
11. In ShowStringDoc.cpp, replace the Class Wizard version of CShowStringDoc::OnToolsOptions() with the OnToolsOptions() from the old ShowString, which puts up the dialog box.
12. In ShowStringDoc.cpp, add #include "OptionsDialog.h" after the #include statements already present.

Build the application, fix any typos or other simple errors, and then execute it. It should run as before, saying Hello, world! in the center of the view. Convince yourself that the Options dialog box still works and that you have restored all the old functionality. Then resize the application and the view as large as possible, so that when you insert an object it doesn't land on the string. Insert an Excel chart as before, and press Esc to stop editing in place. There you have it: A version of ShowString that is also an ActiveX container. Now it's time to get to work making it a *good* container.

Moving, Resizing, and Tracking

The first task you want to do, even when there is only one item contained in ShowString, is to allow the user to move and resize that item. It makes life simpler for the user if you also provide a *tracker rectangle*, a hashed line around the contained item. This is easy to do with the MFC class CRectTracker.

The first step is to add a member variable to the container item (CShowStringCntrItem) definition in CntrItem.h, to hold the rectangle occupied by this container item. Right-click CShowStringCntrItem in ClassView and choose Add Member Variable. The variable type is CRect, the declaration is m_rect; leave the access public.

m_rect needs to be initialized in a function that is called when the container item is first used and then never again. Whereas view classes have OnInitialUpdate() and document classes have OnNewDocument(), container item classes have no such called-only-once function except the constructor. Initialize the rectangle in the constructor, as shown in Listing 14.23.

Listing 14.23 CntrItem.cpp—Constructor

```
CShowStringCntrItem::CShowStringCntrItem(CShowStringDoc* pContainer)
: COleClientItem(pContainer)
{
    m_rect = CRect(10,10,210,210);
}
```

The numerical values used here are those in the boilerplate OnDraw() provided by AppWizard. Now you need to start using the m rect member variable and setting it. The functions affected are presented in the same order as in the earlier section, CShowStringView. First, change CShowStringView::OnDraw(). Find this line:

```
    m_pSelection->Draw(pDC, CRect(10, 10, 210, 210));
```

Replace it with this:

```
m_pSelection->Draw(pDC, m_pSelection->m_rect);
```

Next, change CShowStringCntrItem::OnGetItemPosition(), which needs to return this rectangle. Take away all the comments and the old hardcoded rectangle (leave the ASSERT_VALID macro call), and add this line:

```
    rPosition = m_rect;
```

The partner function

```
    CShowStringCntrItem::OnChangeItemPosition()
```

is called when the user moves the item. This is where m_rect is changed from the initial value. Remove the comments and add code immediately after the call to the base class function, COleClientItem::OnChangeItemPosition(). The code to add is:

```
    m_rect = rectPos;
    GetDocument()->SetModifiedFlag();
```

```
GetDocument()->UpdateAllViews(NULL);
```

Finally, the new member variable needs to be incorporated into

CShowStringCntrItem::Serialize(). Remove the comments and add lines in the storing and saving blocks so that the function looks like Listing 14.24.

Listing 14.24 CntrItem.cpp—CShowStringCntrItem::Serialize()

```
void CShowStringCntrItem::Serialize(CArchive& ar)
{
	ASSERT_VALID(this);
	// Call base class first to read in COleClientItem data.
	// Because this sets up the m_pDocument pointer returned from
	// CShowStringCntrItem::GetDocument, it is a good idea to call
	// the base class Serialize first.
	COleClientItem::Serialize(ar);
	// now store/retrieve data specific to CShowStringCntrItem
	if (ar.IsStoring())
	{
		ar << m_rect;
	}
	else
	{
		ar >> m_rect;
	}
}
```

Build and execute the application, insert a bitmap, and scribble something in it. Press Esc to cancel editing in place, and your scribble shows up in the top-right corner, next to Hello, world!. Choose Edit, Bitmap Image Object and then Edit. (Choosing Open allows you to edit it in a different window.) Use the resizing handles that appear to drag the image over to the left, and then press Esc to cancel in-place editing. The image is drawn at the new position, as expected. Now for the tracker rectangle. The Microsoft tutorials recommend writing a helper function, SetupTracker(), to handle this. Add these lines to CShowStringView::OnDraw(), just after the

call to m_pSelection->Draw():

```
CRectTracker trackrect;
SetupTracker(m_pSelection,&trackrect);
trackrect.Draw(pDC);
```

The one-line statement after the if was not in brace brackets before; don't forget to add them. The entire if statement should look like this:

```
if (m_pSelection != NULL)
{
        m_pSelection->Draw(pDC, m_pSelection->m_rect);
        CRectTracker trackrect;
        SetupTracker(m_pSelection,&trackrect);
        trackrect.Draw(pDC);
}
```

Add the following public function to ShowStringView.h (inside the class definition):

```
void SetupTracker(CShowStringCntrItem* item, CRectTracker* track);
```

Add the code in Listing 14.25 to ShowStringView.cpp immediately after the destructor.

Listing 14.25 ShowStringView.cpp—CShowStringView::SetupTracker()

```
void CShowStringView::SetupTracker(CShowStringCntrItem* item,
CRectTracker* track)
{
        track->m_rect = item->m_rect;
        if (item == m_pSelection)
        {
                track->m_nStyle |= CRectTracker::resizeInside;
        }
        if (item->GetType() == OT_LINK)
        {
                track->m_nStyle |= CRectTracker::dottedLine;
        }
        else
        {
                track->m_nStyle |= CRectTracker::solidLine;
        }
        if (item->GetItemState() == COleClientItem::openState ||
        item->GetItemState() == COleClientItem::activeUIState)
        {
                track->m_nStyle |= CRectTracker::hatchInside;
        }
}
```

This code first sets the tracker rectangle to the container item rectangle. Then it adds styles to the tracker. The styles available are as follows:

- solidLine—Used for an embedded item.
- dottedLine—Used for a linked item.
- hatchedBorder—Used for an in-place active item.
- resizeInside—Used for a selected item.
- resizeOutside—Used for a selected item.
- hatchInside—Used for an item whose server is open.

This code first compares the pointers to this item and the current selection. If they are the same, this item is selected and it gets resize handles. It's up to you whether these handles go on the inside or the outside. Then this code asks the item whether it is linked (dotted line) or not (solid line.) Finally, it adds hatching to active items.

Build and execute the application, and try it out. You still cannot edit the contained item by double-clicking it; choose Edit from the cascading menu added at the bottom of the Edit menu. You can't move and resize an inactive object, but if you activate it, you can resize it while active. Also, when you press Esc, the inactive object is drawn at its new position.

Handling Multiple Objects and Object Selection

The next step is to catch mouse clicks and double-clicks so that the item can be resized, moved, and activated more easily. This involves testing to see whether a click is on a contained item.

Hit Testing

You need to write a helper function that returns a pointer to the contained item that the user clicked, or NULL if the user clicked an area of the view that has no contained item. This function runs through all the items contained in the document. Add the code in Listing 14.26 to ShowStringView.cpp immediately after the destructor.

Listing 14.26 ShowStringView.cpp—CShowStringView::SetupTracker()

```
CShowStringCntrItem* CShowStringView::HitTest(CPoint point)
{
    CShowStringDoc* pDoc = GetDocument();
    CShowStringCntrItem* pHitItem = NULL;
    POSITION pos = pDoc->GetStartPosition();
    while (pos)
    {
        CShowStringCntrItem* pCurrentItem =
        (CShowStringCntrItem*) pDoc->GetNextClientItem(pos);
        if ( pCurrentItem->m_rect.PtInRect(point) )
```

```
        {
                pHitItem = pCurrentItem;
        }
    }
    return pHitItem;
}
```

This function is given a CPoint that describes the point on the screen where the user clicked. Each container item has a rectangle, m_rect, as you saw earlier, and the CRect class has a member function called PtInRect() that takes a CPoint and returns TRUE if the point is in the rectangle or FALSE if it is not. This code simply loops through the items in this document, using the OLE document member function GetNextClientItem(), and calls PtInRect() for each. What happens if there are several items in the container, and the user clicks at a point where two or more overlap? The one on top is selected. That's because GetStartPosition() returns a pointer to the bottom item, and GetNextClientItem() works its way up through the items. If two items cover the spot where the user clicked, pHitItem is set to the lower one first, and then on a later iteration of the while loop, it is set to the higher one. The pointer to the higher item is returned.

Drawing Multiple Items

While that code to loop through all the items is still fresh in your mind, why not fix CShowStringView::OnDraw() so it draws all the items? Leave all the code that draws the string, and replace the code in Listing 14.27 with that in Listing 14.28.

Listing 14.27 ShowStringView.cpp—Lines in OnDraw() to Replace

```
// Draw the selection at an arbitrary position. This code should
// be removed once your real drawing code is implemented. This
// position corresponds exactly to the rectangle returned by
// CShowStringCntrItem, to give the effect of in-place editing.
// TODO: remove this code when final draw code is complete.
if (m_pSelection == NULL)
{
    POSITION pos = pDoc->GetStartPosition();
    m_pSelection = (CShowStringCntrItem*)pDoc-> GetNextClientItem(pos);
}
if (m_pSelection != NULL)
{
    m_pSelection->Draw(pDC, m_pSelection->m_rect);
    CRectTracker trackrect;
```

```
        SetupTracker(m_pSelection,&trackrect);
        trackrect.Draw(pDC);
}
```

Listing 14.28 ShowStringView.cpp—New Lines in OnDraw()

```
POSITION pos = pDoc->GetStartPosition();
while (pos)
{
        CShowStringCntrItem* pCurrentItem =
        (CShowStringCntrItem*) pDoc->GetNextClientItem(pos);
        pCurrentItem->Draw(pDC, pCurrentItem->m_rect);
        if (pCurrentItem == m_pSelection )
        {
                CRectTracker trackrect;
                SetupTracker(pCurrentItem,&trackrect);
                trackrect.Draw(pDC);
        }
}
```

Now each item is drawn, starting from the bottom and working up, and if it is selected, it gets a tracker rectangle.

Handling Single Clicks

When the user clicks the client area of the application, a WM_LBUTTONDOWN message is sent. This message should be caught by the view. Right-click CShowStringView in ClassView, and choose Add Windows Message Handler from the shortcut menu. Click WM_LBUTTONDOWN in the New Windows Messages/Events box on the left (see Figure 14.10), and then click Add and Edit to add a handler function and edit the code immediately.

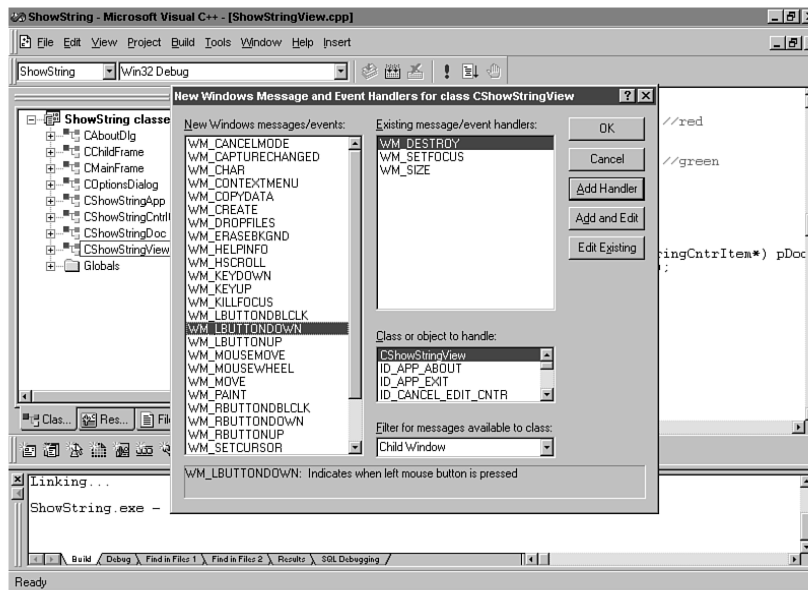


FIG. 14.10 Add a function to handle left mouse button clicks.

Add the code in Listing 14.29 to the empty OnLButtonDown() that Add Windows Message Handler generated.

Listing 14.29 ShowStringView.cpp—

CShowStringView::OnLButtonDown()

```
void CShowStringView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CShowStringCntrItem* pHitItem = HitTest(point);
    SetSelection(pHitItem);
    if (pHitItem == NULL)
    return;
    CRectTracker track;
    SetupTracker(pHitItem, &track);
    UpdateWindow();
    if (track.Track(this,point))
    {
        Invalidate();
        pHitItem->m_rect = track.m_rect;
        GetDocument()->SetModifiedFlag();
    }
}
```

This code determines which item has been selected and sets it. (SetSelection() isn't written yet.) Then, if something has been selected, it draws a tracker rectangle around it and calls CRectTracker::Track(), which allows the user to resize the rectangle. After the resizing, the item is sized to match the tracker rectangle and is redrawn. SetSelection() is pretty straightforward. Add the definition of this public member function to the header file, ShowStringView.h, and the code in Listing 14.30 to ShowStringView.cpp.

Listing 14.30 ShowStringView.cpp—CShowStringView::SetSelection()

```
void CShowStringView::SetSelection(CShowStringCntrItem* item)
{
    // if an item is being edited in place, close it
    if ( item == NULL || item != m_pSelection)
    {
        COleClientItem* pActive =
        GetDocument()->GetInPlaceActiveItem(this);
        if (pActive != NULL && pActive != item)
        {
            pActive->Close();
        }
    }
    Invalidate();
    m_pSelection = item;
}
```

When the selection is changed, any item that is being edited in place should be closed. SetSelection() checks that the item passed in represents a change, and then gets the active object from the document and closes that object. Then it calls for a redraw and sets m_pSelection. Build and execute ShowString, insert an object, and press Esc to stop in-place editing. Click and drag to move the inactive object, and insert another. You should see something like Figure 14.11. Notice the resizing handles around the bitmap, indicating that it is selected.

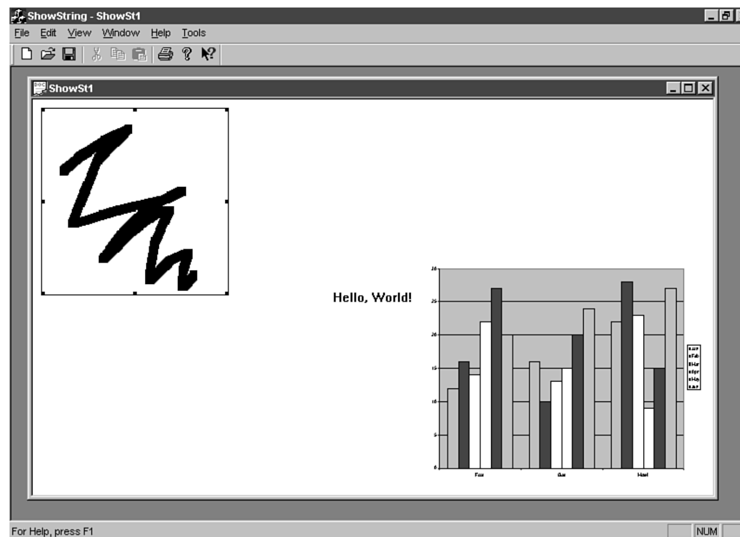


FIG. 14.11 ShowString can now hold multiple items, and the user can move and resize them intuitively.

You might have noticed that the cursor doesn't change as you move or resize. That's because you didn't tell it to. Luckily, it's easy to tell it this: CRectTracker has a SetCursor() member function, and all you need to do is call it when a WM_SETCURSOR message is sent. Again, it should be the view that catches this message; right-click CShowStringView in ClassView, and choose Add Windows Message Handler from the shortcut menu. Click WM_SETCURSOR in the New Windows Messages/Events box on the left; then click Add and Edit to add a handler function and edit the code immediately. Add the code in Listing 14.31 to the empty function that was generated for you.

Listing 14.31 ShowStringView.cpp—CShowStringView::OnSetCursor()

```
BOOL CShowStringView::OnSetCursor(CWnd* pWnd, UINT nHitTest,
UINT message)
{
        if (pWnd == this && m_pSelection != NULL)
        {
                CRectTracker track;
                SetupTracker(m_pSelection, &track);
                if (track.SetCursor(this, nHitTest))
                {
                        return TRUE;
                }
        }
        return CView::OnSetCursor(pWnd, nHitTest, message);
}
```

This code does nothing unless the cursor change involves this view and there is a selection. It gives the tracking rectangle's SetCursor() function a chance to change the cursor because the tracking object knows where the rectangle is and whether the cursor is over a boundary or sizing handle. If SetCursor() didn't change the cursor, this code lets the base class handle it. Build and execute ShowString, and you should see cursors that give you feedback as you move and resize.

Handling Double-Clicks

When a user double-clicks a contained item, the *primary verb* should be called. For most objects, the primary verb is to Edit in place, but for some, such as sound files, it is Play. Arrange as before for CShowStringView to catch the WM_LBUTTONDBLCLK message, and add the code in Listing 14.32 to the new function.

Listing 14.32 ShowStringView.cpp—CShowStringView::OnLButtonDblClk()

```
void CShowStringView::OnLButtonDblClk(UINT nFlags, CPoint point)
{
    OnLButtonDown(nFlags, point);
    if( m_pSelection)
    {
        if (GetKeyState(VK_CONTROL) < 0)
        {
            m_pSelection->DoVerb(OLEIVERB_OPEN, this);
        }
        else
        {
            m_pSelection->DoVerb(OLEIVERB_PRIMARY, this);
        }
    }
    CView::OnLButtonDblClk(nFlags, point);
}
```

First, this function handles the fact that this item has been clicked; calling OnLButtonDown() draws the tracker rectangle, sets m_pSelection, and so on. Then, if the user holds down Ctrl while double-clicking, the item is opened; otherwise, the primary verb is called. Finally, the base class function is called. Build and execute ShowString and try double-clicking. Insert an object, press Esc to stop editing it, move it, resize it, and double-click it to edit in place.

Implementing Drag and Drop

The last step to make ShowString a completely up-to-date ActiveX container application is to implement drag and drop. The user should be able to grab a contained item and drag it out of the container, or hold down Ctrl while dragging to drag out a copy and leave the original behind. The user should also be able to drag items from elsewhere and drop them into this container just as though they had been inserted through the Clipboard. In other words, the container should operate as a *drag source* and a *drop target*.

Implementing a Drag Source

Because CShowStringCntrItem inherits from COleClientItem, implementing a drag source is really easy. By clicking a contained object, edit these lines at the end of CShowStringView::OnLButtonDown() so that it resembles Listing 14.33. The new lines are in bold type.

Listing 14.33 CShowStringView::OnLButtonDown()—Implementing a Drag Source

```
void CShowStringView::OnLButtonDown(UINT nFlags, CPoint point)
{
	CShowStringCntrItem* pHitItem = HitTest(point);
	SetSelection(pHitItem);
	if (pHitItem == NULL)
	return;
	CRectTracker track;
	SetupTracker(pHitItem, &track);
	UpdateWindow();
	if (track.HitTest(point) == CRectTracker::hitMiddle)
	{
		CRect rect = pHitItem->m_rect;
		CClientDC dc(this);
		OnPrepareDC(&dc);
		dc.LPtoDP(&rect); // convert logical rect to device rect
		rect.NormalizeRect();
		CPoint newpoint = point - rect.TopLeft();
		DROPEFFECT dropEffect = pHitItem->DoDragDrop(rect, newpoint);
		if (dropEffect == DROPEFFECT_MOVE)
		{
```

```
                Invalidate();
                if (pHitItem == m_pSelection)
                {
                        m_pSelection = NULL;
                }
                pHitItem->Delete();
            }
        }
        else
        {
            if (track.Track(this,point))
            {
                Invalidate();
                pHitItem->m_rect = track.m_rect;
                GetDocument()->SetModifiedFlag();
            }
        }
}
```

This code first confirms that the mouse click was inside the tracking rectangle, rather than on the sizing border. It sets up a temporary CRect object that will be passed to DoDragDrop() after some coordinate scheme conversions are complete. The first conversion is from logical to device units, and is accomplished with a call to CDC::LPtoDP(). In order to call this function, the new code must create a temporary device context based on the CShowStringView for which OnLButtonDown() is being called. Having converted rect to device units, the new code normalizes it and calculates the point within the rectangle where the user clicked. Then the new code calls the DoDragDrop() member function of CShowStringCntrItem, inherited from COleClientItem and not overridden. It passes in the converted rect and the offset of the click. If DoDragDrop() returns DROPEFFECT_MOVE, the item was moved and needs to be deleted. The code to handle a drop, which is not yet written, will create a new container item and set it as the current selection. This means that if the object was dropped elsewhere in the container, the current selection will no longer be equal to the hit item. If these two pointers are still equal, the object must have been dragged away. If it was dragged away, this code sets m_pSelection to NULL. In either case, pHitItem should be deleted. Build and execute ShowString, insert a new object, press Esc to stop editing in place, and then drag the inactive object to an ActiveX container application such as Microsoft Excel. You can also try dragging to the desktop. Be sure to try dragging an object down to the taskbar and pausing over the icon of a minimized container application, and then waiting while the application is restored so that you can drop the object.

Implementing a Drop Target

It is harder to make ShowString a drop target (it could hardly be easier). If you dragged a contained item out of ShowString and dropped it into another container, try dragging that item back into ShowString. The cursor changes to a circle with a slash through it, meaning "you can't drop that here." In this section, you make the necessary code changes that allow you to drop it there after all.

You need to register your view as a place where items can be dropped. Next, you need to handle the following four events that can occur:

- An item might be dragged across the boundaries of your view. This action will require a cursor change or other indication you will take the item.
- In the view, the item will be dragged around within your boundaries, and you should give the user feedback about that process.
- That item might be dragged out of the window again, having just passed over your view on the way to its final destination.
- The user may drop the item in your view.

Registering the View as a Drop Target

To register the view as a drop target, add a COleDropTarget member variable to the view. In ShowStringView.h, add this line to the class definition:

```
COleDropTarget m_droptarget;
```

To handle registration, override OnCreate() for the view, which is called when the view is

created. Arrange for CShowStringView to catch the WM_CREATE message. Add the code in Listing 14.34 to the empty function generated for you.

Listing 14.34 ShowStringView.cpp—CShowStringView::OnCreate()

```
int CShowStringView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CView::OnCreate(lpCreateStruct) == -1)
		return -1;
	if (m_droptarget.Register(this))
	{
		return 0;
	}
	else
	{
		return -1;
	}
}
```

OnCreate() returns 0 if everything is going well and -1 if the window should be destroyed. This code calls the base class function and then uses COleDropTarget::Register() to register this view as a place to drop items.

Setting Up Function Skeletons and Adding Member Variables

The four events that happen in your view correspond to four virtual functions you must override:

OnDragEnter(), OnDragOver(), OnDragLeave(), and OnDrop(). Right-click

CShowStringView in ClassView and choose Add Virtual Function to add overrides of these functions. Highlight OnDragEnter() in the New Virtual Functions list, click Add Handler, and repeat for the other three functions.

OnDragEnter() sets up a *focus rectangle* that shows the user where the item would go if it were dropped here. This is maintained and drawn by OnDragOver(). But first, a number of member variables related to the focus rectangle must be added to CShowStringView. Add these lines to ShowStringView.h, in the public section:

```
CPoint m_dragpoint;
CSize m_dragsize;
CSize m_dragoffset;
```

A data object contains a great deal of information about itself, in various formats. There is, of course, the actual data as text, *device independent bitmap (DIB)*, or whatever other format is appropriate. But there is also information about the object itself. If you request data in the Object Descriptor format, you can find out the size of the item and where on the item the user originally clicked, and the offset from the mouse to the upper-left corner of the item. These formats are generally referred to as *Clipboard formats* because they were originally used for Cut and Paste via the Clipboard.

To ask for this information, call the data object's GetGlobalData() member function, passing it a parameter that means "Object Descriptor, please." Rather than build this parameter from a string every time, you build it once and store it in a static member of the class. When a class has a static member variable, every instance of the class looks at the same memory location to see that variable. It is initialized (and memory is allocated for it) once, outside the class.

Add this line to ShowStringView.h:

```
static CLIPFORMAT m_cfObjectDescriptorFormat;
```

In ShowStringView.cpp, just before the first function, add these lines:

```
CLIPFORMAT CShowStringView::m_cfObjectDescriptorFormat =
(CLIPFORMAT) ::RegisterClipboardFormat("Object Descriptor");
```

This makes a CLIPFORMAT from the string "Object Descriptor" and saves it in the static member variable for all instances of this class to use. Using a static member variable speeds up dragging over your view.

Your view doesn't accept any and all items that are dropped on it. Add a BOOL member variable to the view that indicates whether it accepts the item that is now being dragged over it:

```
BOOL m_OKtodrop;
```

There is one last member variable to add to CShowStringView. As the item is dragged across the view, a focus rectangle is repeatedly drawn and erased. Add another BOOL member variable that tracks the status of the focus rectangle:

```
BOOL m_FocusRectangleDrawn;
```

Initialize m_FocusRectangleDrawn, in the view constructor, to FALSE:

```
CShowStringView::CShowStringView()
{
    m_pSelection = NULL;
    m_FocusRectangleDrawn = FALSE;
}
```

OnDragEnter()

OnDragEnter() is called when the user first drags an item over the boundary of the view. It sets up the focus rectangle and then calls OnDragOver(). As the item continues to move, OnDragOver() is called repeatedly until the user drags the item out of the view or drops it in the view. The overall structure of OnDragEnter() is shown in Listing 14.35.

Listing 14.35 ShowStringView.cpp—CShowStringView::OnDragEnter()

```
DROPEFFECT CShowStringView::OnDragEnter(COleDataObject* pDataObject,
DWORD dwKeyState, CPoint point)
{
    ASSERT(!m_FocusRectangleDrawn);
    // check that the data object can be dropped in this view
    // set dragsize and dragoffset with call to GetGlobalData
    // convert sizes with a scratch dc
    // hand off to OnDragOver
    return OnDragOver(pDataObject, dwKeyState, point);
}
```

First, check that whatever pDataObject carries is something from which you can make a COleClientItem (and therefore a CShowsStringCntrItem). If not, the object cannot be dropped here, and you return DROPEFFECT_NONE, as shown in Listing 14.36.

Listing 14.36 ShowStringView.cpp—Can the Object Be Dropped?

```
// check that the data object can be dropped in this view
m_OKtodrop = FALSE;
if (!COleClientItem::CanCreateFromData(pDataObject))
    return DROPEFFECT_NONE;
m_OKtodrop = TRUE;
```

Now the weird stuff starts. The GetGlobalData() member function of the data item that is being dragged into this view is called to get the object descriptor information mentioned earlier. It returns a handle of a global memory block. Then the SDK function GlobalLock() is called to convert the handle into a pointer to the first byte of the block and to prevent any other object from allocating the block. This is cast to a pointer to an object descriptor structure (the undyingly curious can check about 2,000 lines into oleidl.h, in the \Program Files\Microsoft Visual Studio\VC98\Include folder for most installations, to see the members of this structure) so that the sizel and pointl elements can be used to fill the \m_dragsize and m_dragoffset member variables.

Finally, GlobalUnlock() reverses the effects of GlobalLock(), making the block accessible to others, and GlobalFree() frees the memory. It ends up looking like Listing 14.37.

Listing 14.37 ShowStringView.cpp—Set dragsize and dragoffset

```
// set dragsize and dragoffset with call to GetGlobalData
HGLOBAL hObjectDescriptor = pDataObject->GetGlobalData(
m_cfObjectDescriptorFormat);
if (hObjectDescriptor)
{
    LPOBJECTDESCRIPTOR pObjectDescriptor =
    (LPOBJECTDESCRIPTOR) GlobalLock(hObjectDescriptor);
    ASSERT(pObjectDescriptor);
    m_dragsize.cx = (int) pObjectDescriptor->sizel.cx;
    m_dragsize.cy = (int) pObjectDescriptor->sizel.cy;
    m_dragoffset.cx = (int) pObjectDescriptor->pointl.x;
    m_dragoffset.cy = (int) pObjectDescriptor->pointl.y;
    GlobalUnlock(hObjectDescriptor);
    GlobalFree(hObjectDescriptor);
}
```

```
else
{
    m_dragsize = CSize(0,0);
    m_dragoffset = CSize(0,0);
}
```

For some ActiveX operations, global memory is too small—imagine trying to transfer a 40MB file through global memory! There is a more general function than GetGlobalData(), called (not surprisingly) GetData(), which can transfer the data through a variety of storage medium choices. Because the object descriptors are small, asking for them in global memory is a sensible approach.

If the call to GetGlobalData() didn't work, set both member variables to zero by zero rectangles. Next, convert those rectangles from OLE coordinates (which are device independent) to pixels:

```
// convert sizes with a scratch dc
    CClientDC dc(NULL);
    dc.HIMETRICtoDP(&m_dragsize);
        dc.HIMETRICtoDP(&m_dragoffset);
```

HIMETRICtoDP() is a very useful function that happens to be a member of CClientDC, which inherits from the familiar CDC of Chapter 5, "Drawing on the Screen." You create an instance of CClientDC just so you can call the function.

OnDragEnter() closes with a call to OnDragOver(), so that's the next function to write.

OnDragOver()

This function returns a DROPEFFECT. As you saw earlier in the "Implementing a Drag Source" section, if you return DROPEFFECT_MOVE, the source deletes the item from itself. Returning DROPEFFECT_NONE rejects the copy. It is OnDragOver() that deals with preparing to accept or reject a drop. The overall structure of the function looks like this:

```
DROPEFFECT CShowStringView::OnDragOver(COleDataObject* pDataObject,
DWORD dwKeyState, CPoint point)
{
    // return if dropping is already rejected
    // determine drop effect according to keys depressed
    // adjust focus rectangle
}
```

First, check to see whether OnDragEnter() or an earlier call to OnDragOver() already rejected this possible drop:

```
// return if dropping is already rejected
    if (!m_OKtodrop)
    {
        return DROPEFFECT_NONE;
    }
```

Next, look at the keys that the user is holding down now, available in the parameter passed to this function, dwKeyState. The code you need to add (see Listing 14.38) is straightforward.

Listing 14.38 ShowStringView.cpp—Determine the Drop Effect

```
// determine drop effect according to keys depressed
DROPEFFECT dropeffect = DROPEFFECT_NONE;
if ((dwKeyState & (MK_CONTROL|MK_SHIFT) )== (MK_CONTROL|MK_SHIFT))
{
    // Ctrl+Shift force a link
    dropeffect = DROPEFFECT_LINK;
}
else if ((dwKeyState & MK_CONTROL) == MK_CONTROL)
{
    // Ctrl forces a copy
    dropeffect = DROPEFFECT_COPY;
}
else if ((dwKeyState & MK_ALT) == MK_ALT)
{
    // Alt forces a move
    dropeffect = DROPEFFECT_MOVE;
}
else
{
    // default is to move
    dropeffect = DROPEFFECT_MOVE;
}
```

If the item has moved since the last time OnDragOver() was called, the focus rectangle has to be erased and redrawn at the new location. Because the focus rectangle is a simple XOR of the colors, drawing it a second time in the same place removes it. The code to adjust the focus rectangle is in Listing 14.39.

Listing 14.39 ShowStringView.cpp—Adjust the Focus Rectangle

```
// adjust focus rectangle
point -= m_dragoffset;
if (point == m_dragpoint)
{
    return dropeffect;
}
CClientDC dc(this);
if (m_FocusRectangleDrawn)
{
    dc.DrawFocusRect(CRect(m_dragpoint, m_dragsize));
    m_FocusRectangleDrawn = FALSE;
}
if (dropeffect != DROPEFFECT_NONE)
{
    dc.DrawFocusRect(CRect(point, m_dragsize));
    m_dragpoint = point;
    m_FocusRectangleDrawn = TRUE;
}
```

To test whether the focus rectangle should be redrawn, this code adjusts the point where the user clicked by the offset into the item to determine the top-left corner of the item. It can then compare that location to the top-left corner of the focus rectangle. If they are the same, there is no need to redraw it. If they are different, the focus rectangle might need to be erased.

Finally, replace the return statement that was generated for you with one that returns the

calculated DROPEFFECT:

```
return dropeffect;
```

OnDragLeave()

Sometimes a user drags an item right over your view and out the other side. OnDragLeave() just tidies up a little by removing the focus rectangle, as shown in Listing 14.40.

Listing 14.40 ShowStringView.cpp—ShowStringView::OnDragLeave()

```
void CShowStringView::OnDragLeave()
{
    CClientDC dc(this);
    if (m_FocusRectangleDrawn)
    {
        dc.DrawFocusRect(CRect(m_dragpoint, m_dragsize));
        m_FocusRectangleDrawn = FALSE;
    }
}
```

OnDragDrop()

If the user lets go of an item that is being dragged over ShowString, the item lands in the container and OnDragDrop() is called. The overall structure is in Listing 14.41.

Listing 14.41 ShowStringView.cpp—Structure of OnDrop()

```
BOOL CShowStringView::OnDrop(COleDataObject* pDataObject,
DROPEFFECT dropEffect, CPoint point)
{
    ASSERT_VALID(this);
    // remove focus rectangle
    // paste in the data object
    // adjust the item dimensions, and make it the current selection
    // update views and set modified flag
    return TRUE;
}
```

Removing the focus rectangle is simple, as shown in Listing 14.42.

Listing 14.42 ShowStringView.cpp—Removing the Focus Rectangle

```
// remove focus rectangle
```

```
CClientDC dc(this);
if (m_FocusRectangleDrawn)
{
        dc.DrawFocusRect(CRect(m_dragpoint, m_dragsize));
        m_FocusRectangleDrawn = FALSE;
}
```

Next, create a new item to hold the data object, as shown in Listing 14.43. Note the use of the bitwise and (&) to test for a link.

Listing 14.43 ShowStringView.cpp—Paste the Data Object

```
// paste the data object
CShowStringDoc* pDoc = GetDocument();
CShowStringCntrItem* pNewItem = new CShowStringCntrItem(pDoc);
ASSERT_VALID(pNewItem);
if (dropEffect & DROPEFFECT_LINK)
{
        pNewItem->CreateLinkFromData(pDataObject);
}
else
{
        pNewItem->CreateFromData(pDataObject);
}
ASSERT_VALID(pNewItem);
```

The size of the container item needs to be set, as shown in Listing 14.44.

Listing 14.44 ShowStringView.cpp—Adjust Item Dimensions

```
        // adjust the item dimensions, and make it the current selection
        CSize size;
        pNewItem->GetExtent(&size, pNewItem->GetDrawAspect());
        dc.HIMETRICtoDP(&size);
        point -= m_dragoffset;
        pNewItem->m_rect = CRect(point,size);
        m_pSelection = pNewItem;
```

Notice that this code adjusts the place where the user drops the item (point) by m_dragoffset, the coordinates into the item where the user clicked originally. Finally, make sure the document is saved on exit, because pasting in a new container item changes it, and redraw the view:

```
// update views and set modified flag
pDoc->SetModifiedFlag();
pDoc->UpdateAllViews(NULL);
return TRUE;
```

This function always returns TRUE because there is no error checking at the moment that might require a return of FALSE. Notice, however, that most problems have been prevented; for example, if the data object cannot be used to create a container item, the DROPEFFECT would have been set to DROPEFFECT_NONE in OnDragEnter() and this code would never have been called. You can be confident this code works.

Testing the Drag Target

All the confidence in the world is no substitute for testing. Build and execute ShowString, and try dragging something into it. To test both the drag source and drop target aspects at once, drag something out and then drag it back in. Now this is starting to become a really useful container. There's only one task left to do.

Deleting an Object

You can remove an object from your container by dragging it away somewhere, but it makes sense to implement deleting in a more obvious and direct way. The menu item generally used for this is Edit, Delete, so you start by adding this item to the IDR_SHOWSTTYPE menu before the Insert New Object item. Don't let Developer Studio set the ID to ID_EDIT_DELETE; instead, change it to ID_EDIT_CLEAR, the traditional resource ID for the command that deletes a contained object. Move to another menu item and then return to Edit, Delete, and you see that the prompt has been filled in for you as Erase the selection\nErase automatically. The view needs to handle this command, so add a message handler as you have done throughout this chapter. Follow these steps:

1. Right-click CShowStringView in ClassView and choose Add Windows Message Handler.

2. Choose ID_EDIT_CLEAR from the Class or Object to Handle drop-down box at the lower right.

3. Choose COMMAND from the New Windows Messages/Events box that appears when you click the ID_EDIT_CLEAR box.

4. Click Add Handler.

5. Click OK to accept the suggested name.

6. Choose UPDATE_COMMAND_UI from the New Windows Messages/Events box and click Add Handler again.

7. Accept the suggested name.

8. Click OK on the large dialog to complete the process.

The code for these two handlers is very simple. Because the update handler is simpler, add code to it first:

```
void CShowStringView::OnUpdateEditClear(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_pSelection != NULL);
}
```

If there is a current selection, it can be deleted. If there is not a current selection, the menu item is disabled (grayed). The code to handle the command isn't much longer: it's in Listing 14.45.

Listing 14.45 ShowStringView.cpp—CShowStringView::OnEditClear()

```
void CShowStringView::OnEditClear()
{
    if (m_pSelection)
    {
        m_pSelection->Delete();
        m_pSelection = NULL;
        GetDocument()->SetModifiedFlag();
        GetDocument()->UpdateAllViews(NULL);
    }
}
```

This code checks that there is a selection (even though the menu item is grayed when there is no selection) and then deletes it, sets it to NULL so that there is no longer a selection, makes sure the document is marked as modified so that the user is prompted to save it when exiting, and gets the view redrawn without the deleted object.

Build and execute ShowString, insert something, and delete it by choosing Edit, Delete. Now it's an intuitive container that does what you expect a container to do.

REFERENCES

1. Kate Gregory, “Special Edition Using Visual C++6”,QUE Publications,1998.
2. Chris-II an William –II and Murray-III “The Complete Reference Visual C++6” Tata Mc Graw Publications.
3. Bronson, “Visual C++”, Thomson Asia Publications.
4. Hitesh Sanghavi, “Programming with Visual C++”, Vikas Publications.
5. Kanitkar, “Visual C++ Programming”, BPB Publications.